Fyne Golang

Haylem Candelario Bauzá. Habana Cuba 2025. Documento Libre.

Getting Started

Using the Fyne toolkit to build cross platform applications is very simple but does require some tools to be installed before you can begin. If your computer is set up for development with Go then the following steps may not be required, but we advise reading the tips for your operating system just in case. If later steps in this tutorial fail then you should re-visit the prerequisites below.

Prerequisites

Fyne requires 3 basic elements to be present, the Go tools (at least version 1.19), a C compiler (to connect with system graphics drivers) and a system graphics driver. The instructions vary depending on your operating system, choose the appropriate tab below for installation instructions.

Note that these steps are just required for development - your Fyne applications will not require any setup or dependency installation for end users!

Supported Platforms see: wiki/Supported-Platforms

You will need to install Go, GCC and the graphics library header files using your package manager. Choose the command corresponding to your distribution:

- **Debian, Ubuntu and Raspberry Pi OS:** `sudo apt-get install golang gcc libgl1-mesa-dev xorg-dev libxkbcommon-dev`
- **Fedora:** `sudo dnf install golang golang-misc gcc libXcursor-devel libXrandr-devel mesa-libGL-devel libXi-devel libXinerama-devel libXxf86vm-devel libxkbcommon-devel wayland-devel`
- **Arch Linux:** `sudo pacman -S go xorg-server-devel libxcursor libxrandr libxinerama libxi libxkbcommon`
- **Solus:** `sudo eopkg it -c system.devel golang mesalib-devel libxrandr-devel libxcursor-devel libxi-devel libxinerama-devel libxkbcommon-devel`
- **openSUSE:** `sudo zypper install go gcc libXcursor-devel libXrandr-devel Mesa-libGL-devel libXi-devel libXinerama-devel libXxf86vm-devel libxkbcommon-devel`
- **Void Linux:** `sudo xbps-install -S go base-devel xorg-server-devel libXrandr-devel libXcursor-devel libXinerama-devel libXxf86vm-devel libxkbcommon-devel wayland-devel`
- **Alpine Linux** `sudo apk add go gcc libxcursor-dev libxrandr-dev libxinerama-dev libxi-dev linux-headers mesa-dev libxkbcommon-dev wayland-dev`
- **NixOS** `nix-shell -p libGL pkg-config xorg.libX11.dev xorg.libXcursor xorg.libXi xorg.libXinerama xorg.libXrandr xorg.libXxf86vm libxkbcommon wayland`

Downloading

Since Go 1.16 you will need to set up the module before you can use the package.

Run the following command and replace `MODULE_NAME` with your preferred module name (this should be called in a new folder specific for your application).

```
$ mkdir myapp
$ cd myapp
$ go mod init MODULE_NAME
```

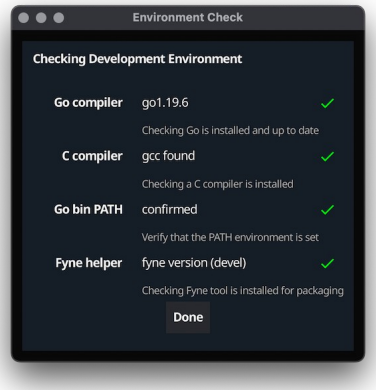
You now need to download the Fyne module and helper tool. This will be done using the following commands:

```
$ go get fyne.io/fyne/v2@latest
$ go install fyne.io/tools/cmd/fyne@latest
```

If you are unsure of how Go modules work, consider reading Tutorial: Create a Go module.

Check your installation

Before coding an app or running an example you can check your install using the Fyne Setup tool. Simply download the right app for your computer from the link and run it, you should see something like the following screen:



If there are any problems with your installation see the troubleshooting section for hints.

Run the demo

If you want to see the Fyne toolkit in action before you start to code your own application, you can see our demo app running on your computer by executing:

```
$ go run fyne.io/demo@latest
```

Please note that the first run has to compile some C-code and can thus take longer than usual. Subsequent builds reuse the cache and will be much faster.

Installing

If you want to, you can also install the demo using the following command (requires Go 1.16 or later):

```
$ go install fyne.io/demo@latest
```

If your `GOBIN` environment has been added to path (should be by default on macOS and Windows), you can then run the demo:

```
$ demo
```

And that's all there is to it! Now you can write your own Fyne application in your IDE of choice. If you want to see some Fyne code in action then you can read your first application.

Creating your first Fyne app

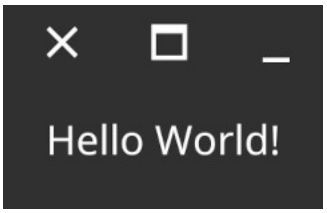
Having completed the steps in the getting started document you're ready to build your first app. To illustrate the process we will build a simple hello world application.

A simple app starts by creating an app instance with app.New() and then opening a window with app.NewWindow(). Then a widget tree is defined that is set as the main content with SetContent() on a window. The app UI is then shown by calling ShowAndRun() on the window.

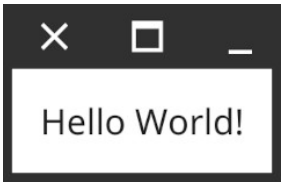
```
package main

import (
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/widget"
)

func main() {
        a := app.New()
        w := a.NewWindow("Hello World")

        w.SetContent(widget.NewLabel("Hello World!"))
        w.ShowAndRun()
}
```

The code above can be built using the command `go build .` and then executed either by running the `hello` command or by double clicking the icon. You could also bypass the compiling step and just run the code directly using `go run .`.

Either approach will show a window that looks just like this:



If you prefer a light theme then just set the environment variable `FYNE_THEME=light` and you'll get:



That's all there is to getting started. To learn more you can read the full API documentation.

Run Fyne Demo

If you want to see the Fyne toolkit in action before you start to code your own application, you can see our demo app.

Running

If you want to, you can run the demo directly using the following command (requires Go 1.16 or later):

```
go run fyne.io/demo@latest
```

For earlier versions of Go, you need to use the following command instead:

```
go run fyne.io/demo
```

By browsing the different tabs of the app you can see all the features of Fyne toolkit.

Installing

It is possible to install the app as a graphical application like all others on your computer. We have the helpful `fyne` tool to do this for you. First you will need to install the tool:

```
go install fyne.io/tools/cmd/fyne@latest
```

After that you can simply package and install the demo app:

```
fyne install fyne.io/demo

(This may require admin privileges on your system, and that in turn may require some path changes to include the fyne and go executables.)
```

After this step you can find “Fyne Demo” in your app launcher.

Exploring the code

If you are interested in any of the features you should check out the source code or join one of the community channels.

Application and RunLoop

For a GUI application to work it needs to run an event loop (sometimes called a runloop) that processes user interactions and drawing events. In Fyne this is started using the `App.Run()` or `Window.ShowAndRun()` functions. One of these must be called from the end of your setup code in the `main()` function.

An application should only have one runloop and so you should only call `Run()` once in your code. Calling it a second time will cause errors.

```
package main

import (
        "fmt"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Hello")
        myWindow.SetContent(widget.NewLabel("Hello"))

        myWindow.Show()
        myApp.Run()
        tidyUp()
}
```

```
func tidyUp() {
        fmt.Println("Exited")
}
```

For desktop runtimes an app can be quit directly by calling `App.Quit()` (mobile apps do not support this) - normally not needed in developer code. An application will also quit once all the windows are closed. See also that functions executed after `Run()` will not be called until the application exits.

Updating Content in your GUI

Having completed the hello world tutorial or other examples you will have created a basic user interface. In this page we see how the content of a GUI can be updated from your code.

The first step is to assign the widget you want to update to a variable. In the hello world tutorial we passed `widget.NewLabel` directly into `SetContent()`, to update it we change that to two different lines, such as:

```
        clock := widget.NewLabel("")
        w.SetContent(clock)
```

Once the content has been assigned to a variable we can call functions like `SetText("new text")`. For our example we will set the content of our label to the current time, with the help of `Time.Format`.

```
        formatted := time.Now().Format("Time: 03:04:05")
        clock.SetText(formatted)
```

That is all we need to do to change content of a visible item (see below for the full code). However, we can go further and update content on a regular basis.

Running in the background

Most applications will need to have processes that run in the background, for example downloading data or responding to events. To simulate this we will extend the code above to run every second.

Like with most go code we can create a goroutine (using the `go` keyword) and run our code there. If we move the text update code to a new function it can be called on initial display as well as on a timer for regular updating. By combining a goroutine and the `time.Tick` inside a for loop we can update the label every second.

```
        go func() {
                for range time.Tick(time.Second) {
                        updateTime(clock)
                }
        }()
```

It is important to place this code before `ShowAndRun` or `Run` calls because they will not return until the application closes. With all of this together the code will run and update the user interface each second, creating a basic clock widget. The full code is as follows:

```
package main

import (
        "time"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/widget"
)

func updateTime(clock *widget.Label) {
        formatted := time.Now().Format("Time: 03:04:05")
        clock.SetText(formatted)
}

func main() {
        a := app.New()
        w := a.NewWindow("Clock")

        clock := widget.NewLabel("")
        updateTime(clock)
```

```
        w.SetContent(clock)
        go func() {
                for range time.Tick(time.Second) {
                        updateTime(clock)
                }
        }()
        w.ShowAndRun()
}
```

Window Handling

Windows are created using `App.NewWindow()` and need to be shown using the `Show()` function. The helper method `ShowAndRun()` on `fyne.Window` allows you to show your window and run the application at the same time.

By default a window will be the right size to show its content by checking the `MinSize()` function (more on that in later examples). You can set a larger size by calling the `Window.Resize()` method. Into this is passed a `fyne.Size` which contains a width and height using device independent pixels (meaning that it will be the same across different devices), for example to make a window square by default we could:

```
        w.Resize(fyne.NewSize(100, 100))
```

Be aware that the desktop environment may have constraints that cause windows to be smaller than requested. Mobile devices will typically ignore this as they are only displayed at full-screen.

If you wish to show a second window you must only call the `Show()` function. It can also be helpful to split `Window.Show()` from `App.Run()` if you want to open multiple windows when your application starts. The example below shows how to load two windows when starting.

```
package main

import (
        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/widget"
)

func main() {
        a := app.New()
        w := a.NewWindow("Hello World")

        w.SetContent(widget.NewLabel("Hello World!"))
        w.Show()

        w2 := a.NewWindow("Larger")
        w2.SetContent(widget.NewLabel("More content"))
        w2.Resize(fyne.NewSize(100, 100))
        w2.Show()

        a.Run()
}
```

The above application will exit when both windows are closed. If your app is arranged so one window is main and the others are accessory views you can set one window to be “master” so that the app exits if that window is closed. To do this use the `SetMaster()` function on `Window`.

Windows can be created at any time, we could change the code above so that the content of the second window (`w2`) is a button that opens a new window. You could also load windows from more complex workflows, but be careful because new windows will normally appear above the current active content.

```
        w2.SetContent(widget.NewButton("Open new", func() {
                w3 := a.NewWindow("Third")
                w3.SetContent(widget.NewLabel("Third"))
                w3.Show()
        }))
```

Using Goroutines

Since Fyne v2.6.0 all Fyne events and callbacks happen on the same goroutine - this allows for higher performance and much better thread safety than any previous version.

With this change applications that wish to use goroutines (to do background updates for example) have a little more work to do to get the same safety and performance as internal widgets.

Updating from a goroutine

Any time your app invokes Fyne APIs from a goroutine your code created, you should use the `fyne.Do` function to tell Fyne that you want to queue updates to the rendered application, for example with a `canvas.Text` called `output` you may want to update the time each second. The key code there would be:

```
        fyne.Do(func() {
                output.Text = time.Now().Format(time.TimeOnly)
                output.Refresh()
        })
```

Such an update would occur in a goroutine that may be started from your main func. You can see a complete clock example app below:

```
package main

import (
        "image/color"
        "time"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
)

func main() {
        a := app.New()
        w := a.NewWindow("Hello")

        output := canvas.NewText(time.Now().Format(time.TimeOnly), color.NRGBA{G: 0xff,
A: 0xff})
        output.TextStyle.Monospace = true
        output.TextSize = 32
        w.SetContent(output)

        go func() {
                ticker := time.NewTicker(time.Second)
                for range ticker.C {
                        fyne.Do(func() {
                                output.Text = time.Now().Format(time.TimeOnly)
                                output.Refresh()
                        })
                }
        }()
        w.ShowAndRun()
}
```

Waiting for the code to run

When calling `fyne.Do` the code will execute between when you request it and the next frame to draw - which could be up to 15ms later. Some times it is necessary to know when that process has finished. For example if you are updating an image buffer and do not want to start again until after it is processed for the next visual update.

To do this another function, `fyne.DoAndWait` is useful. This takes the same parameter as the `Do` version above, but it will not return until the update is complete. This means the next line in your app will happen after the user interface is updated - very useful if you want to avoid potential issues with accessing shared resources concurrently.

Testing Graphical Apps

Part of a good test suite is being able to quickly write tests and run them on a regular basis. Fyne's API is designed to make testing applications easy. By separating component logic from its rendering definition we can load applications without actually displaying them and test the functionality completely.

Example

We can demonstrate unit testing by extending our Hello World app to include space for users to input their name to be greeted. We start by updating the user interface to have two elements, a `Label` for the greeting and an `Entry` for the name input. We display them, one above another, using `container.NewVBox` (a vertical box container). The updated user interface code will look as follows:

```
func makeUI() (*widget.Label, *widget.Entry) {
        return widget.NewLabel("Hello world!"),
                widget.NewEntry()
}

func main() {
        a := app.New()
        w := a.NewWindow("Hello Person")

        w.SetContent(container.NewVBox(makeUI()))
        w.ShowAndRun()
}
```

To test this input behaviour we create a new file (with a name ending `_test.go` to mark it as tests) that defines a `TestGreeter` function.

```
package main

import (
        "testing"
)

func TestGreeting(t *testing.T) {
}
```

We can add an initial test that verifies the initial state, to do this we test the `Text` field of the `Label` that is returned from `makeUI` and error the test if it is not correct. Add the following code to your test method:

```
        out, in := makeUI()

        if out.Text != "Hello world!" {
                t.Error("Incorrect initial greeting")
        }
```

This test will pass - next we add to the test to validate the greeter. We use the Fyne `fyne.io/fyne/v2/test` package which assists in test scenarios, calling `test.Type` to simulate user input. The following test code will check that the output updates when the user's name is input (be sure to add the import as well):

```
        test.Type(in, "Andy")
        if out.Text != "Hello Andy!" {
                t.Error("Incorrect user greeting")
        }
```

You can run all of these tests using `go test .` - just like any other tests. Doing so you will now see a failure - because we did not add the greeter logic. Update the `makeUI` function to the following code:

```
func makeUI() (*widget.Label, *widget.Entry) {
        out := widget.NewLabel("Hello world!")
        in := widget.NewEntry()

        in.OnChanged = func(content string) {
                out.SetText("Hello " + content + "!")
        }
        return out, in
```

```
}
```

Doing so you will see that the tests now pass. You can also run the full application (using `go run .`) and see the greeting update as you enter a name in the `Entry` field. Notice also that these tests all run without displaying a window or stealing your mouse - this is another benefit of the Fyne unit testing setup.

Packaging for Desktop

Packaging a graphical app for distribution can be complex. Graphical applications typically have icons and metadata associated with them as well as specific formats required to integrate with each environment. Windows executables need embedded icons, macOS apps are bundles and with Linux there are various metadata files that should get installed. What a hassle!

Thankfully the “fyne” app has a “package” command that can handle this automatically. Just specifying the target OS and any required metadata (such as icon) will generate the appropriate package. The icon conversion will be done automatically for .icns or .ico so just provide a .png file :). All you need is to have the application already built for the target platform…

```
go install fyne.io/tools/cmd/fyne@latest
fyne package -os darwin -icon myapp.png
```

If you’re using an older version of Go (<1.16), you should install fyne using `go get`

```
go get fyne.io/tools/cmd/fyne
fyne package -os darwin -icon myapp.png
```

Will create myapp.app, a complete bundle structure for distribution to macOS users. You could then build the linux and Windows versions too…

```
fyne package -os linux -icon myapp.png
fyne package -os windows -icon myapp.png
```

These commands will create:

- myapp.tar.gz that contains a folder structure starting at usr/local/ that a Linux user could expand to the root of their system.
- myapp.exe (after the second build, which is required for a windows package) will have the icon and app metadata embedded.

If you just want to install the desktop app on your computer then you can make use of the helpful install subcommand. For example to install your current application system wide you could simply execute the following:

```
fyne install -icon myapp.png
```

All of these commands also support a default icon file of `Icon.png` so that you can avoid typing the parameter for each execution. Since Fyne 2.1 there is also a metadata file where you can set default options for your project.

Of course you can still run your applications using the standard Go tools if you prefer.

Mobile Packaging

Your Fyne app code will work out of the box as mobile apps, just as it did for desktop. However it is a little more complex to package the code for distribution. This page will explore the process to do just that to get your app on iOS and Android.

Firstly you will need some more development tools installed for mobile packaging to complete. For Android builds you must have the Android SDK and NDK installed with appropriate environment set up so that the tools (such as `adb`) can be found on the command line. To build iOS apps you will need Xcode installed on your macOS computer as well as the command line tools optional package.

Once you have a working development environment the packaging is simple. To build an application for Android and iOS the following commands will do everything for you. Be sure to have a unique application identifier as it is unwise to change these after your first release.

```
fyne package -os android -appID com.example.myapp -icon mobileIcon.png
fyne package -os ios -appID com.example.myapp -icon mobileIcon.pngAfter these commands have completed (which may take some time on first compilation) you will see two new files in your directory, myapp.apk and myapp.app. You will see that the latter has the same name as a darwin application bundle - don’t get them confused as they will not work on the other platform.
```

To install the android app on your phone or a simulator simply call:

```
adb install myapp.apk
```

For iOS to install on device open Xcode and choose the “Devices and Simulators” menu item within the “Window” menu. Then find your phone and drag the `myapp.app` icon onto your app list.

If you want to install on a simulator make sure to package your application using `iossimulator` vs `ios`

```
fyne package -os iossimulator -appID com.example.myapp -icon mobileIcon.png
```

Afterwards you can use the command line tools as follows:

```
xcrun simctl install booted myapp.app
```

Run in a Browser

Fyne applications can also be run over the web using a standard web browser! A web app created with Fyne will bundle a WebAssembly binary which will run in most modern browsers

To prepare your app to be used over the web we use the “fyne” cli app again, which has a “serve” command for quick testing

```
go install fyne.io/tools/cmd/fyne@latest
fyne serve
```

You will see, after a few moments, that a web server has been started on port :8080. Just open your web browser to https://localhost:8080 and you can use your app!

Packaging for web distribution

The `fyne serve` command is great for local testing, but just like other platforms you’ll want to be able to distribute your app as well. To prepare the files for upload just use the `fyne package` command like with regular Packaging.

```
fyne package -os web
```

Demo

You can see a Fyne app in action to test on any of your devices by visiting demo.fyne.io.

Limitations

As of release v2.5.0 the web driver is fully supported but is not yet 100% complete, so your app may not be able to use the following features:

- File open/save dialog
- Storage of Documents

These issues are being worked on and will be resolved in a future release.

App Metadata

Since release v2.1.0 of the `fyne` command we support a metadata file that allows you to store information about your application in the repository. This file is optional, but can help to avoid having to remember specific build parameters for each package and release command.

Basic configuration

The file should be named `FyneApp.toml` in the directory where you run the `fyne` command (this is normally the `main` package). The contents of the file are as follows:

```
Website = "https://example.com"

[Details]
Icon = "Icon.png"
Name = "My App"
ID = "com.example.app"
Version = "1.0.0"
Build = 1
```

The top portion of the file is metadata that will be used if you upload your app to the Apps listing page, so it is optional.

The `Details` table contains data about your application that are used in the release process by other app stores and operating systems.

The `fyne` tool will use this file if it is found, many mandatory command parameters are not required if the metadata is present. You can still override these values by using command line parameters.

Linux & BSD configuration

For Linux and BSD builds there is an optional table called `LinuxAndBSD`. This table contains additional parameters for a “desktop entry” configuration file of the Fyne app. All parameters are optional, but when present they will be used by the `fyne` tool (in addition to parameters from the `Details` table).

The contents of this section is as follows (with example data):

```
[LinuxAndBSD]
  GenericName = "Web Browser"
  Categories = ["Network"]
  Comment = "View sites on the Internet"
  Keywords = ["browser", "web"]
  ExecParams = "-x 42"
```

Hint: For instructions on how to define these parameters correctly, please see the Desktop Entry Specification from freedesktop.org.

Distributing to App Stores

Packaging a graphical app as described in the Packaging page provides a file or bundle that could be directly shared or distributed. However signing and uploading to app stores and market places is an additional step that requires platform-specific configuration, which we will cover in this page.

In each of these steps we will use a new tool that is part of the fyne command line utilities. The `fyne release` step handles the signing and preparation for each store, but the parameters vary per-platform, which we see below.

macOS App Store (since fyne 1.4.2)

Prerequisites:

- Apple mac running macOS and Xcode
- Apple Developer account
- Mac App Store application certificate
- Mac App Store installer certificate
- Apple Transporter app from App Store

1. Set up your app / version ready for a build to be uploaded at AppStore Connect.

2. Bundle the completed app for release:

```
$ fyne release -appID com.example.myapp -appVersion 1.0 -appBuild 1 -category games
```

3. Drag the `.pkg` onto Transporter and tap “Deliver”.

4. Go to back to the AppStore Connect website, choose your build for the release and submit for review.

Google Play Store (Android)

Prerequisites:

- Google Play Console account
- distribution keystore (creation instructions in android docs)

1. Set up your app / version ready for build to be uploaded at Google Play Console. Turn off “Play app signing” option as we manage it ourselves.

2. Bundle the completed app for release:

```
$ fyne release -os android -appID com.example.myapp -appVersion 1.0 -appBuild 1
```

3. Drag the `.apk` file into the build drop zone on the app version page in Play Console

4. Start rollout of new version.

iOS App Store (since fyne 1.4.1)

Prerequisites:

- Apple mac running macOS and Xcode
- Apple Developer account
- iOS App Store distribution certificate
- Apple Transporter app from App Store

1. Set up your app / version ready for a build to be uploaded at AppStore Connect.

2. Bundle the completed app for release:

```
$ fyne release -os ios -appID com.example.myapp -appVersion 1.0 -appBuild 1
```

3. Drag the `.ipa` onto Transporter and tap “Deliver”.

4. Go to back to the AppStore Connect website, choose your build for the release and submit for review.

Compiling for different platforms

Cross compiling with Go is designed to be simple - we just set the environment variable `GOOS` for the target Operating System (and `GOARCH` if targeting a different architecture). Unfortunately when using native graphics calls the use of CGo in Fyne makes this a little harder.

Compiling from a development computer

To cross-compile a Fyne application you will also have to set `CGO_ENABLED=1` which tells go to enable the C compiler (this is normally turned off when the target platform is different to the current system). Doing so unfortunately means that you must have a C compiler for the target platform that you are going to compile for. After installing the appropriate compilers you will also need to set the `CC` environment variable to tell Go which compiler to use.

There are many ways to install the required tools - and different tools that can be used. The configuration recommended by the Fyne developers is:

GOOS (target)	CC	provider	download	notes
`darwin`	`o32-clang`	osxcross	from github.com	You will also need to install the macOS SDK (instructions at the download link)
`windows`	`x86_64-w64-mingw64-gcc`	mingw64	package manager	For macOS use homebrew
`linux`	`gcc` or `x86_64-linux-musl-gcc`	gcc or musl-cross	cygwin or package manager	musl-cross is available from homebrew to provide the linux gcc. You will also need to install X11 and mesa headers for compilation.

With the environment variables above set you should be able to compile in the usual manner. If further errors occur it is likely to be due to missing packages. Some target platforms require additional libraries or headers to be installed for the compilation to succeed.

Using a virtual environment

As a Linux system is able to cross compile to macOS and Windows easily it can be simpler to use a virtualised environment when you are not developing from Linux. Docker images are a useful tool for a complex build configuration and this works for Fyne as well. There are different tools that can be used. The tool recommended by the Fyne developers is fyne-cross. It has been inspired by xgo and uses a docker image built on top of the golang-cross image, that includes the MinGW compiler for windows, and a macOS SDK, along with the Fyne requirements.

fyne-cross allows to build binaries and create distribution packages for the following targets:

GOOS	GOARCH
darwin	amd64
darwin	386
linux	amd64
linux	386
linux	arm64
linux	arm
windows	amd64
windows	386
android	amd64
android	386
android	arm64
android	arm
ios	
freebsd	amd64
freebsd	arm64

Note: iOS compilation is supported only on darwin hosts.

Requirements

- go >= 1.13
- docker

Installation

You can install `fyne-cross` using the following command (requires Go 1.16 or later):

```
go install github.com/fyne-io/fyne-cross@latest
```

For earlier versions of Go, you need to use the following command instead:

```
go get github.com/fyne-io/fyne-cross
```

Usage

```
fyne-cross <command> [options]

The commands are:

        darwin        Build and package a fyne application for the darwin OS
        linux         Build and package a fyne application for the linux OS
        windows       Build and package a fyne application for the windows OS
        android       Build and package a fyne application for the android OS
        ios           Build and package a fyne application for the iOS OS
        freebsd       Build and package a fyne application for the freebsd OS
        version       Print the fyne-cross version information

Use "fyne-cross <command> -help" for more information about a command.
```

Wildcards

The `arch` flag support wildcards in case want to compile against all supported GOARCH for a specified GOOS

Example:

```
fyne-cross windows -arch=*
```

is equivalent to

```
fyne-cross windows -arch=amd64,386
```

Example

The example below cross compile and package the fyne examples application

```
git clone https://github.com/fyne-io/examples.git
cd examples
```

Compile and package the main example app

```
fyne-cross linux
```

> Note: by default fyne-cross will compile the package into the current dir.
>
> The command above is equivalent to: `fyne-cross linux .`

Compile and package a particular example app

```
fyne-cross linux -output bugs ./cmd/bugs
```

Canvas and CanvasObject

In Fyne a `Canvas` is the area within which an application is drawn. Each window has a canvas which you can access with `Window.Canvas()` but usually you will find functions on `Window` that avoid accessing the canvas.

Everything that can be drawn in Fyne is a type of `CanvasObject`. The example here opens a new window and then shows different types of primitive graphical element by setting the content of the window canvas. There are many ways that each type of object can be customised as shown with the text and circle examples.

As well as changing the content shown using `Canvas.SetContent()` it is possible to change the content that is currently visible. If, for example, you change the `FillColour` of a rectangle you can request a refresh of this existing component using `rect.Refresh()`.

```
package main

import (
        "image/color"
        "time"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Canvas")
        myCanvas := myWindow.Canvas()

        blue := color.NRGBA{R: 0, G: 0, B: 180, A: 255}
        rect := canvas.NewRectangle(blue)
        myCanvas.SetContent(rect)

        go func() {
                time.Sleep(time.Second)
                green := color.NRGBA{R: 0, G: 180, B: 0, A: 255}
                rect.FillColor = green
                rect.Refresh()
        }()

        myWindow.Resize(fyne.NewSize(100, 100))
        myWindow.ShowAndRun()
}
```

We can draw many different drawing elements in the same way, such as circle and text.

```
func setContentToText(c fyne.Canvas) {
        green := color.NRGBA{R: 0, G: 180, B: 0, A: 255}
        text := canvas.NewText("Text", green)
        text.TextStyle.Bold = true
        c.SetContent(text)
}

func setContentToCircle(c fyne.Canvas) {
        red := color.NRGBA{R: 0xff, G: 0x33, B: 0x33, A: 0xff}
        circle := canvas.NewCircle(color.White)
        circle.StrokeWidth = 4
        circle.StrokeColor = red
        c.SetContent(circle)
}
```

Widget

A `fyne.Widget` is a special type of canvas object that has interactive elements associated with it. In widgets the logic is separate from the way that it looks (also called the `WidgetRenderer`).

Widgets are also types of `CanvasObject` and so we can set the content of our window to a single widget. See how we create a new `widget.Entry` and set it as the content of the window in this example.

```
package main

import (
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Widget")

        myWindow.SetContent(widget.NewEntry())
        myWindow.ShowAndRun()
}
```

Container and Layouts

In the previous example we saw how to set a `CanvasObject` to the content of a `Canvas`, but it is not very useful to only show one visual element. To show more than one item we use the `Container` type.

As the `fyne.Container` also is a `fyne.CanvasObject`, we can set it to be the content of a `fyne.Canvas`. In this example we create 3 text objects and then place them in a container using the `container.NewWithoutLayout()` function. As there is no layout set we can move the elements around like you see with `text2.Move()`.

```
package main

import (
        "image/color"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
        "fyne.io/fyne/v2/container"
        //"fyne.io/fyne/v2/layout"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Container")
        green := color.NRGBA{R: 0, G: 180, B: 0, A: 255}

        text1 := canvas.NewText("Hello", green)
        text2 := canvas.NewText("There", green)
        text2.Move(fyne.NewPos(20, 20))
        content := container.NewWithoutLayout(text1, text2)
        // content := container.New(layout.NewGridLayout(2), text1, text2)

        myWindow.SetContent(content)
        myWindow.ShowAndRun()
}
```

A `fyne.Layout` implements a method for organising items within a container. By uncommenting the `container.New()` line in this example you alter the container to use a grid layout with 2 columns. Run this code and try resizing the window to see how the layout automatically configures the contents of the window. Notice also that the manual position of `text2` is ignored by the layout code.

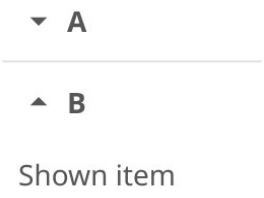
To see more you can check out the `Layout list`.

Widget List

Standard Widgets (in `widget` package)

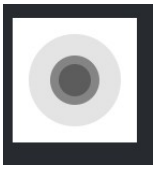
Accordion

Accordion displays a list of AccordionItems. Each item is represented by a button that reveals a detailed view when tapped.



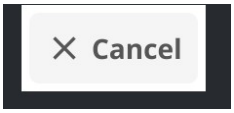
Activity

Display an animated activity indicator.



Button

Button widget has a text label and icon, both are optional.

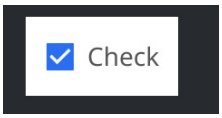


Card

Card widget groups elements with a header and subheader, all are optional.

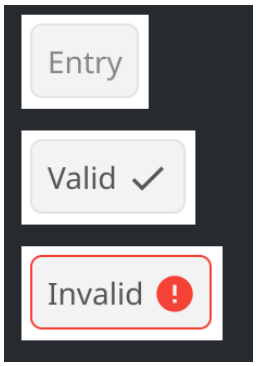
Check

Check widget has a text label and a checked (or unchecked) icon.

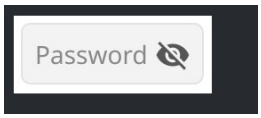


Entry

Entry widget allows simple text to be input when focused.

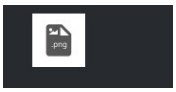


PasswordEntry widget hides text input and adds a button to display the text.



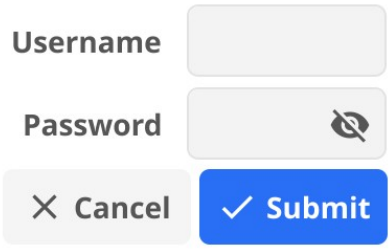
FileIcon

FileIcon provides helpful standard icons for various types of file. It displays the type of file as an indicator icon and shows the extension of the file type.



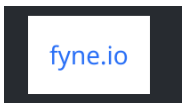
Form

Form widget is two column grid where each row has a label and a widget (usually an input). The last row of the grid will contain the appropriate form control buttons if any should be shown.



Hyperlink

Hyperlink widget is a text component with appropriate padding and layout. When clicked, the URL opens in your default web browser.



Icon

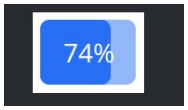
Icon widget is a basic image component that load’s its resource to match the theme.

Label

Label widget is a label component with appropriate padding and layout.

Progress bar

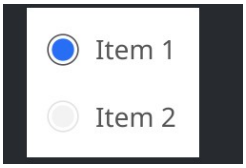
ProgressBar widget creates a horizontal panel that indicates progress.



ProgressBarInfinite widget creates a horizontal panel that indicates waiting indefinitely An infinite progress bar loops 0% -> 100% repeatedly until Stop() is called.

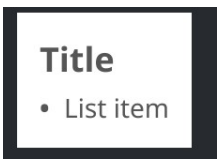
RadioGroup

RadioGroup widget has a list of text labels and radio check icons next to each.



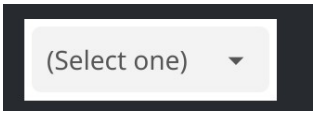
RichText

RichText widget is a text component that shows various styles and embedded objects. It supports markdown parsing to construct the widget with ease.



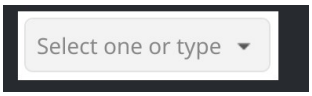
Select

Select widget has a list of options, with the current one shown, and triggers an event function when clicked.



SelectEntry

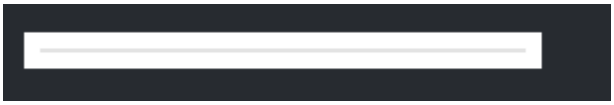
Select entry widget adds an editable component to the select widget. Users can select an option or enter their own value.



parator

Separator widget shows a dividing line between other elements

.



Slider

Slider if a widget that can slide between two fixed values.

TextGrid

TextGrid is a monospaced grid of characters. This is designed to be used by a text editor, code preview or terminal emulator.

Toolbar

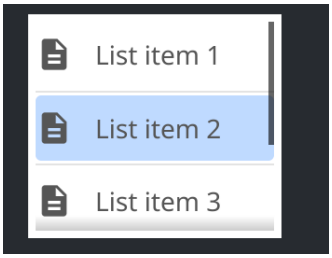
Toolbar widget creates a horizontal list of tool buttons.

Collection Widgets (in `widget` package)

Collection widgets provide advanced caching functionality to provide high performance rendering of massive data. This does lead to a more complex constructor, but is a good balance for the outcome it enables. Each of these widgets uses a series of callbacks, the minimum set is defined by their constructor function, which includes the data size, the creation of template items that can be re-used and finally the function that applies data to a widget as it is about to be added to the display.

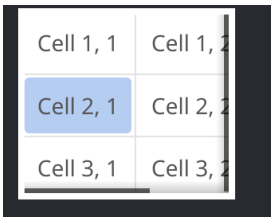
List

List provides a high performance vertical scroll of many sub-items.



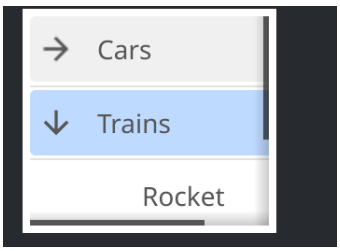
Table

Table provides a high performance scrolled two dimensional display of many sub-items.



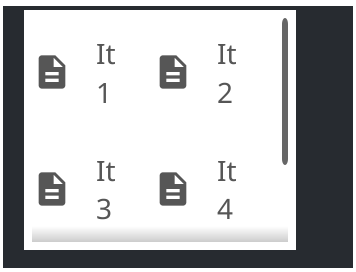
Tree

Tree provides a high performance vertical scroll of items that can be expanded to reveal child elements..



GridWrap

GridWrap is a collection widget that display each child item at the same size and wraps them to new lines as required.

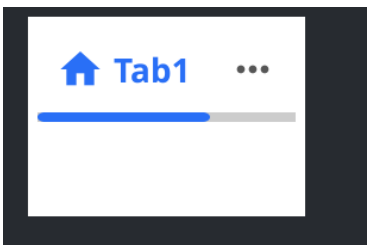


Container Widgets (in `container` package)

Container widgets are like regular containers but they provide some additional functionality.

AppTabs

AppTabs widget allows switching visible content from a list of TabItems. Each item is represented by a button at the top of the widget.

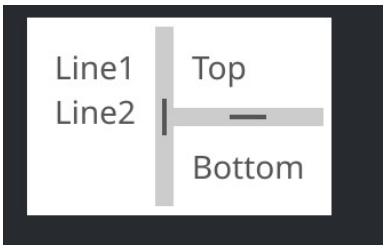


Scroll

ScrollContainer defines a container that is smaller than the Content.

Split

SplitContainer defines a container whose size is split between two children.



Layout List

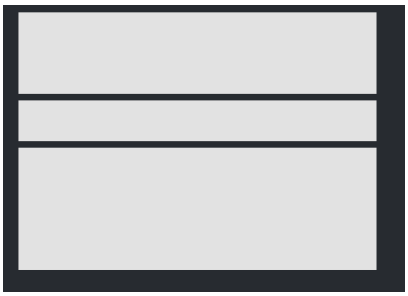
Standard Layouts

Horizontal Box (HBox)

Horizontal Box arranges items in a horizontal row. Every element will have the same height (the height of the tallest item in the container) and objects will be left-aligned at their minimum width.

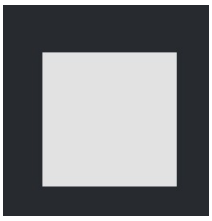
Vertical Box (VBox)

Vertical Box arranges items in a vertical column. Every element will have the same width (the width of the widest item in the container) and objects will be top-aligned at their minimum height.



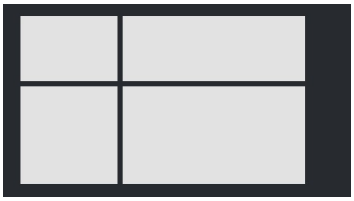
Center

Center layout positions all container elements in the center of the container. Every object will be set to it’s minimum size.



Form

Form layout arranges items in pairs where the first column is at minimum width. This is normally useful for labelling elements in a form, where the label is in the first column and the item it describes is in the second. You should always add an even number of elements to a form layout.

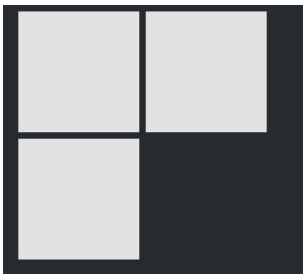


Grid

Grid layout arranges items equally in the available space. A number of columns is specified, with objects being positioned horizontally until the number of columns is reached at which point a new row is started. All objects have the same size, that is width divided by column total and the height will be total height divided by the number of rows required. Minus padding.

GridWrap

GridWrap layout arranges all items to flow along a row, wrapping to a new row if there is insufficient space. All objects will be set to the same size, which is the size passed to the layout. This layout may not respect item MinSize to manage this uniform layout. Often used in file managers or image thumbnail lists.



Border

Border layout supports positioning of items at the outside of available space. The border is passed pointers to the objects for (top, left, bottom, right). All items in the container that are not positioned on a border will fill the remaining space.

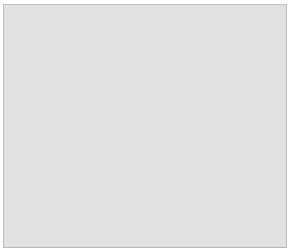
Stack (was Max)

Stack layout positions all container elements to fill the available space. The objects will all be full-sized and drawn in the order they were added to the container (last-most is on top).

Padded

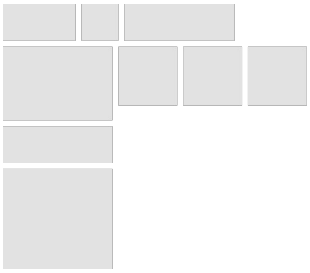
Padded layout positions all container elements to fill the available space but with a small padding around the outside. The size of the padding is theme specific. The objects will all be drawn in the order they were added to the container (last-most is on top).

ou can also specify each site separately using
`CustomPadded` layout.



Combining Layouts

It is possible to build up more complex application structures by using multiple layouts. Multiple containers that each have their own layout can be nested to create complete user interface arrangements using only the standard layouts listed above. For example a horizontal box for a header, a vertical box for a left side file panel and a grid wrap layout in the content area - all inside a container using a border layout can build the result illustrated below.

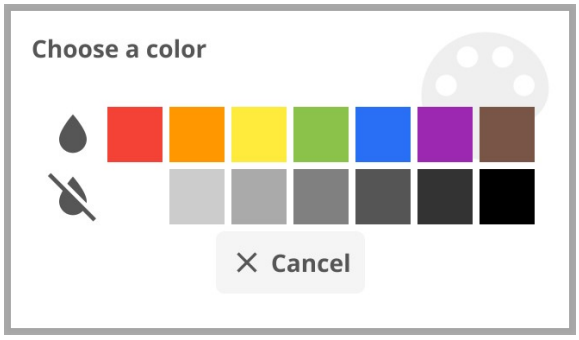


Dialog List

Standard Dialogs

Color

Allow users to pick a colour from a standard set (or any color in advanced mode).

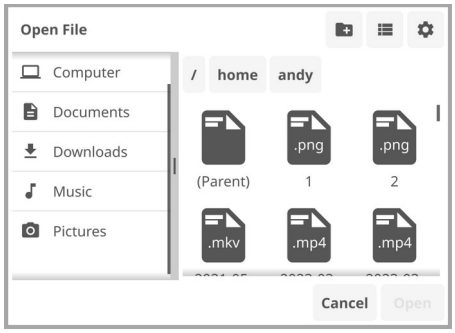


Confirm

Ask for conformation of an action.

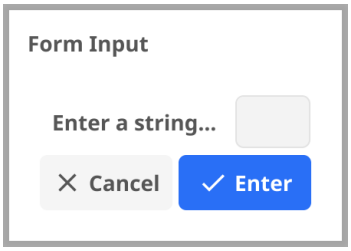
FileOpen

Present this to ask user to choose a file to use inside the app. The actual dialog displayed will depend on the current operating system.



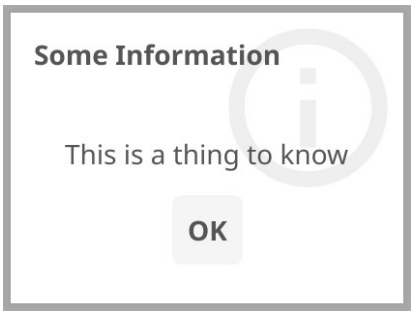
Form

Get various input elements in a dialog, with validation.



Information

A simple way to present some information to the app user.



Custom

Present any content inside a dialog container.

Theme Icons

Each of the following icons is available via the `theme` package as a function. For example `theme.InfoIcon()`.

The icons are also available via their source icon name by using the `ThemeIconName` with the `Icon` method on a struct implementing `fyne.Theme`. For example `theme.Icon(theme.IconNameInfo)`.

List

-

- DocumentCreateIcon
- DocumentIcon
- DocumentPrintIcon
- DocumentSaveIcon
- DownloadIcon
- ErrorIcon
- FileApplicationIcon
- FileAudioIcon
- FileIcon
- FileImageIcon
- FileTextIcon
- FileVideoIcon
- FolderIcon
- FolderIcon
- FolderNewIcon
- FolderOpenIcon
- GridIcon
- HelpIcon
- HistoryIcon
- HomeIcon
- InfoIcon
- ListIcon
- LoginIcon
- LogoutIcon
- MailAttachmentIcon
- MailComposeIcon
- MailForwardIcon
- MailReplyAllIcon

- AccountIcon
- ArrowDropDownIcon
- ArrowDropUpIcon
- BrokenImageIcon
- CancelIcon
- CheckButtonIcon
- CheckButtonCheckedI
- ColorAchromaticIcon
- ColorChromaticIcon
- ColorPaletteIcon
- ComputerIcon
- ConfirmIcon

- MailReplyIcon
- MailSendIcon
- MediaFastForwardIco
- MediaFastRewindIcon
- MediaMusicIcon
- MediaPauseIcon
- MediaPhotoIcon
- MediaPlayIcon
- MediaRecordIcon
- MediaReplayIcon
- MediaSkipNextIcon
- MediaSkipPreviousIcc
- MediaStopIcon
- MediaVideoIcon
- MenuExpandIcon

- ViewFullScreenIcor

Conference

- ViewRestoreIcon
- ViewZoomFitIcon
- ViewZoomInIcon
- ViewZoomOutIcon
- VisibilityOffIcon
- VisibilityIcon
- VolumeDownIcon
- VolumeMuteIcon
- VolumeUpIcon
- WarningIcon

- ContentAddIcon
- ContentClearIcon
- ContentCopyIcon
- ContentCutIcon
- ContentPasteIcon
- ContentRedoIcon
- ContentRemoveIcon
- ContentUndoIcon
- DeleteIcon

- MenuIcon
- MoreHorizontalIcon
- MoreVerticalIcon
- MoveDownIcon
- MoveUpIcon
- NavigateBackIcon
- NavigateNextIcon
- QuestionIcon
- RadioButtonCheckedIcon
- RadioButtonIcon
- SearchReplaceIcon
- SearchIcon
- SettingsIcon
- StorageIcon
- UploadIcon

Adding Shortcuts to an App

Shortcuts are common tasks that can be triggered by keyboard combinations or context menus. Shortcuts, much like keyboard events, can be attached to a focused element or registered on the `Canvas` to always be available in a `Window`.

Registering with a Canvas

There are many standard shortcuts defined (such as `fyne.ShortcutCopy`) which are connected to standard keyboard shortcuts and right-click menus. The first step to adding a new `Shortcut` is to define the shortcut. For most uses this will be a keyboard triggered shortcut, which is a desktop extension. To do this we use `desktop.CustomShortcut`, for example to use the Tab key and Control modifier you might do the following:

```
        ctrlTab    := &desktop.CustomShortcut{KeyName: fyne.KeyTab, Modifier:
fyne.KeyModifierControl}
        ctrlAltTab := &desktop.CustomShortcut{KeyName: fyne.KeyTab, Modifier:
fyne.KeyModifierControl | fyne.KeyModifierAlt}
```

Notice that this shortcut can be re-used so you could attach it to menus or other items as well. For this example we want it to be always available, so we register it with our window's `Canvas` as follows:

```
        ctrlTab := &desktop.CustomShortcut{KeyName: fyne.KeyTab, Modifier:
fyne.KeyModifierControl}
        w.Canvas().AddShortcut(ctrlTab, func(shortcut fyne.Shortcut) {
                log.Println("We tapped Ctrl+Tab")
        })
        w.Canvas().AddShortcut(ctrlAltTab, func(shortcut fyne.Shortcut) {
                log.Println("We tapped Ctrl+Alt+Tab")
        })
```

As you can see there are two parts to registering a shortcut in this way - passing the shortcut definition and also a callback function. If the user types the keyboard shortcut then the function will be called and the output printed.

Shortcuts only work in combination with modifier keys. In order to react to keyboard input without modifiers use `canvas.OnTypedRune` or `canvas.OnTypedKey`.

Adding shortcuts to an Entry

It can also be helpful to have a shortcut apply only when the current item is focused. This approach can be used for any focusable widget, and is managed by extending that widget and adding a `TypedShortcut` handler. This is much like adding key handlers, except the value passed in will be a `fyne.Shortcut`.

```
type myEntry struct {
        widget.Entry
}

func (m *myEntry) TypedShortcut(s fyne.Shortcut) {
        if _, ok := s.(*desktop.CustomShortcut); !ok {
                m.Entry.TypedShortcut(s)
                return
        }

        log.Println("Shortcut typed:", s)
}
```

From the excerpt above you can see how a `TypedShortcut` handler might be implemented. Inside this function you should check whether the shortcut is of the custom type used earlier. If the shortcut is a standard one it's a good idea to call the original shortcut handler (if the widget had one). With those checks done you can compare the shortcut with the various types you are handling (if there are multiple).

Using the Preferences API

Storing user configurations and values is a common task for application developers, but implementing it across multiple platforms can be tedious and time-consuming. To make it easier, Fyne has an API for storing values on the filesystem in a clean and understandable way while the complex parts are handled for you.

Lets start with the setup of the API. It is part of the Preferences interface where storage and loading functions exist for values of `bool`, `float64`, `int`, and `string` types, and for lists of each type as well. For each type there are three different functions, one for loading, one loading with a fallback value and lastly, one for storing values. An example of the three functions and their behaviour can be seen below for the `string` type:

```
// String looks up a string value for the key
String(key string) string
// StringWithFallback looks up a string value and returns the given fallback if not found
StringWithFallback(key, fallback string) string
// SetString saves a string value for the given key
SetString(key string, value string)
```

These functions can be accessed through the created application variable and calling the `Preferences()` method on. Please note that it is necessary to create the apps with a unique ID (usually like a reversed url). This means that the application will need to be created using `app.NewWithID()` (or have ID specified in `FyneApp.toml`) to have its own place to store values. It can roughly be used like the example below:

```
a := app.NewWithID("com.example.tutorial.preferences")
[...]
a.Preferences().SetBool("Boolean", true)
a.Preferences().SetFloatList("Constants", []float64{3.142, 2.718})
number := a.Preferences().IntWithFallback("ApplicationLuckyNumber", 21)
expression := a.Preferences().String("RegularExpression")
[...]
```

To show this, we are going to build a simple little app that always closes after a set amount of time. This timeout should be user changeable and applied on the next start of the application.

Let us start by creating a variable called `timeout` that will be used to store time in the form of `time.Duration`.

```
var timeout time.Duration
```

Then we could create a select widget to let the user select the timeout from a couple pre-defined strings and then multiplying the timeout by the number of seconds that the string relates to. Lastly, the `"AppTimeout"` key is used to set the string value to the selected one.

```
timeoutSelector := widget.NewSelect([]string{"10 seconds", "30 seconds", "1 minute"},
func(selected string) {
    switch selected {
    case "10 seconds":
        timeout = 10 * time.Second
    case "30 seconds":
        timeout = 30 * time.Second
    case "1 minute":
        timeout = time.Minute
    }

    a.Preferences().SetString("AppTimeout", selected)
})
```

Now we want to grab the set value and if none exists, we want to have a fallback that sets the timeout to the shortest one possible to save the user time when waiting. This can be done by setting the selected value of `timeoutSelector` to the loaded value or the fallback if that happens to be the case. By doing it this way, the code inside the select widget will run for that specific value.

```
timeoutSelector.SetSelected(a.Preferences().StringWithFallback("AppTimeout", "10
seconds"))
```

The last part will just be to have a function that starts in a separate goroutine and tells the application to quit after the selected timeout.

```
go func() {
    time.Sleep(timeout)
    a.Quit()
}()
```

In the end, the resulting code should look something like this:

```
package main

import (
    "time"

    "fyne.io/fyne/v2/app"
    "fyne.io/fyne/v2/widget"
)

func main() {
    a := app.NewWithID("com.example.tutorial.preferences")
    w := a.NewWindow("Timeout")

    var timeout time.Duration

    timeoutSelector := widget.NewSelect([]string{"10 seconds", "30 seconds", "1 minute"},
func(selected string) {
        switch selected {
        case "10 seconds":
            timeout = 10 * time.Second
        case "30 seconds":
            timeout = 30 * time.Second
        case "1 minute":
            timeout = time.Minute
        }

        a.Preferences().SetString("AppTimeout", selected)
    })

    timeoutSelector.SetSelected(a.Preferences().StringWithFallback("AppTimeout", "10
seconds"))

    go func() {
        time.Sleep(timeout)
        a.Quit()
    }()

    w.SetContent(timeoutSelector)
    w.ShowAndRun()
}
```

Adding app translations

Most apps will want to add translations at some point, and since v2.5.0 Fyne helps to make this really simple. The translation features simply sit in your project as `.json` files and the content can be crowd-sourced by popular platforms like Weblate.

Setup

The Fyne toolkit is already translated and will replace recognised strings in standard locations. To make use of this functionality for your own app strings you make use of the `fyne.io/fyne/v2/lang` package and the helper functions it provides.

The simplest way to prepare for translations is to use the `L` or `Localize` function to mark a string as translatable - if the translation is not found then the string will be used as a fallback.

```
    title := widget.NewLabel(lang.L("My App Title"))
```

In some cases it may be desirable to label a string with a unique ID instead of using the default value - for disambiguation or other reason. In this case you would use `LocalizeKey` or the `X` alias.

```
    title := widget.NewLabel(lang.X("window.title", "My App Window Title"))
```

That may be all you need to know to get started - read on to learn how to create your translation files.

Translation files

Each translation file is a simple JSON file, if you use the `L` form all you need to do is insert the string with it's translation for each language - 1 per file. For example, this may be your app `en.json`:

```
{
  "My App Title": "My App Title"
}
```

Then your French translation could look like:

```
{
  "My App Title": "Titre de mon application"
}
```

Each file can be most easily loaded using the Go `embed` feature - place each of the files in a directory called `translation` and then define them simply as:

```
//go:embed translation
var translations embed.FS
```

Finally you can tell Fyne to load these translations with a single function call between `app.New` and `Run()` in your `main` function:

```
        lang.AddTranslationsFS(translations, "translation")
```

This uses the embedded filesystem and specifies the name of the directory that the files are stored in.

The naming of these translation files should follow the format [prefix].json.

For example:

```
$ tree # at tranlations
.
|-- de.json
|-- en.json
|-- zh.json
```

When your app starts it will display using the translations for the current user's language configuration.

Plurals

In more complex cases the string will change based on the number of items it refers to. For this the `lang.LocalizePlural` function (aliased to `lang.N`) is available.

```
    age := widget.NewLabel(lang.N("{{.Years}} years old", years, map[string]any{"Years":
years}))
```

You can pass data in this way to any of the calls, using template syntax to insert the value. A struct with exported fields, or a map as illustrated above, can be used to insert data.

The translation file will look a little more complex for this - the key has two sub-values called "one" and "other" (some languages may use more keys).

```
{
  "Age": {
    "one": "1 year old",
    "other": "{{.Years}} years old"
  }
}
```

System Tray Menu

Adding a System Tray menu

Since the v2.2.0 release Fyne has built in support for a system tray menu. This feature displays an icon on macOS, Windows and Linux computers and when tapped will pop out a menu as specified by the app.

As this is a desktop specific feature we must first do a runtime check that the app is running in desktop mode. To do this, and get a reference to the desktop features, we do a Go type assertion:

```
        if desk, ok := a.(desktop.App); ok {
...
        }
```

If the `ok` variable is true then we can set up a menu using the standard Fyne menu API that you might have used in `Window.SetMainMenu` before.

```
                m := fyne.NewMenu("MyApp",
                        fyne.NewMenuItem("Show", func() {
                                log.Println("Tapped show")
                        }))
                desk.SetSystemTrayMenu(m)
```

With this code added to the setup of your application you can run the app and see that it shows a Fyne icon in the system tray. When you tap it a menu will appear containing “Show” and “Quit”.

The default icon is the Fyne logo, you can either fix this using app metadata or by setting the app icon in `App.SetIcon` or for system tray directly using `desk.SetSystemTrayIcon`.

Manage window lifecycle

By default a Fyne app will exit when you close all windows and this may not be what you want with a system tray app. To override the behaviour you can use the `Window.SetCloseIntercept` feature to override what happens when a window is closed. In the example below we hide the window instead of closing it by calling `Window.Hide()`. Add this before you show the window for the first time.

```
        w.SetCloseIntercept(func() {
                w.Hide()
        })
```

The benefit of hiding a window is that you can simply show it again using `Window.Show()` which is much more efficient than creating a new window if the same content is needed a second time. We update the menu created earlier to show the window that was hidden above.

```
        fyne.NewMenuItem("Show", func() {
                w.Show()
        })
```

Complete app

That’s all there is to setting up a system tray menu with Fyne! The complete code for this tutorial is as follows.

```
package main

import (
        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/driver/desktop"
        "fyne.io/fyne/v2/widget"
)

func main() {
        a := app.New()
        w := a.NewWindow("SysTray")

        if desk, ok := a.(desktop.App); ok {
                m := fyne.NewMenu("MyApp",
```

```
                        fyne.NewMenuItem("Show", func() {
                                w.Show()
                        }))
                desk.SetSystemTrayMenu(m)
        }

        w.SetContent(widget.NewLabel("Fyne System Tray"))
        w.SetCloseIntercept(func() {
                w.Hide()
        })
        w.ShowAndRun()
}
```

Data Binding

Data binding was introduced in Fyne v2.0.0 and makes it easier to connect many widgets to a data source that will update over time. the `data/binding` package has many helpful bindings that can manage most standard types that will be used in an application. A data binding can be managed using the binding API (for example `NewString`) or it can be connected to an external item of data like (`BindInt(*int)`).

Widgets that support binding typically have a `...WithData` constructor to set up the binding when creating the widget. You can also call `Bind()` and `Unbind()` to manage the data of an existing widget. The following example shows how you can manage a `String` data item that is bound to a simple `Label` widget.

```
package main

import (
        "time"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/data/binding"
        "fyne.io/fyne/v2/widget"
)

func main() {
        a := app.New()
        w := a.NewWindow("Hello")

        str := binding.NewString()
        go func() {
                dots := "....."
                for i := 5; i >= 0; i-- {
                        str.Set("Count down" + dots[:i])
                        time.Sleep(time.Second)
                }
                str.Set("Blast off!")
        }()

        w.SetContent(widget.NewLabelWithData(str))
        w.ShowAndRun()
}
```

You can find out more in the data binding section of this site.

Migrations

In some upgrades the Fyne toolkit may deliver a major feature that is a non-trivial upgrade for larger applications. To allow this to happen without major API changes we have introduced a migrations system. This allows apps to opt in to a new feature or way of working over time, to test the changes carefully and then mark the project as migrated.

It is recommended to stay up to date with migrations within a year of them being introduced, as the toolkit will switch all these features on over time (following deprecation lifecycle).

In v2.6.0 Fyne introduced a new threading model which is the first time we used a migration. If you have not already you should see the details of this feature at Using Goroutines. Once you have adopted this feature follow on to see how a migration is marked as complete.

Opting in to a migration

Migrations are controlled by the new “[Migrations]” section inside `FyneApp.toml`. Each line in this section is a name for the migration followed by a boolean value to switch it on or off.

You can see below an example metadata file where we have turned on the “fyneDo” migration. This should be done once you have updated your code so that any goroutines in your code use the [fyne.Do] method to ensure the code runs on the toolkit’s main goroutine.

```
[Details]
  Icon = "Icon.png"
  Name = "Example"
  ID = "com.example.app"

[Migrations]
  fyneDo = true
```

Testing a migration

It is also possible to test a migration before you mark it as complete - there is a compiler flag that will use this feature for the current build command. For example, to test the fyne.Do migration you could build as follows:

```
go build -tags migrated_fynedo .
```

Compile Options

Build tags

Fyne will typically configure your application appropriately for the target platform by selecting the driver and configuration. The following build tags are supported and can help in your development. For example if you wish to simulate a mobile application whilst running on a desktop computer you could use the following command:

```
go run -tags mobile main.go
```

Tag	Description
`debug`	Show debug information, including visual layout to help understand your app.
`flatpak`	Build with improved support for running inside the Flatpak sandbox on Linux.
`gles`	Force use of embedded OpenGL (GLES) instead of full OpenGL. This is normally controlled by the target device and not normally needed.
`hints`	Display developer hints for improvements or optimisations. Running with `hints` will log when your application does not follow material design or other recommendations.
`mobile`	This tag runs an application in a simulated mobile window. Useful when you want to preview your app on a mobile platform without compiling and installing to the device.
`no_animations`	Disable the non-essential animations in standard widgets and containers.
`no_emoji`	Don't include the embedded emoji font. This will disable emoji in your app but will make the binary smaller.
`no_metadata`	Disable runtime lookup of metadata from FyneApp.toml source files. This is always disabled for release builds.
`no_native_menus`	This flag is specifically for macOS and indicates that the application should not use the macOS native menus. Instead menus will be displayed inside the application window. Most useful for testing an application on macOS to simulate the behavior on Windows or Linux.
`wayland`	Build with support for the Wayland window protocol instead of X11. Only relevant on Linux and BSD systems.

Rectangle

`canvas.Rectangle` is the simplest canvas object in Fyne. It displays a block of the specified colour. You can also set the colour using the `FillColor` field.

In this example the rectangle fills the window as it is the only content element.

```
package main

import (
        "image/color"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
)

func main() {
        myApp := app.New()
        w := myApp.NewWindow("Rectangle")

        rect := canvas.NewRectangle(color.White)
        w.SetContent(rect)

        w.Resize(fyne.NewSize(150, 100))
        w.ShowAndRun()
}
```

Other `fyne.CanvasObject` types have more configuration, let us look next at `canvas.Text`.

Text

`canvas.Text` is used for all text rendering within Fyne. It is created by specifying the text and colour for the text. Text is rendered using the default font, specified by the current theme.

The text object allows certain configuration like the `Alignment` and `TextStyle` field. as illustrated in the example here. To use a monospaced font instead you can specify `fyne.TextStyle{Monospace: true}`.

```
package main

import (
        "image/color"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
)

func main() {
        myApp := app.New()
        w := myApp.NewWindow("Text")

        text := canvas.NewText("Text Object", color.White)
        text.Alignment = fyne.TextAlignTrailing
        text.TextStyle = fyne.TextStyle{Italic: true}
        w.SetContent(text)

        w.ShowAndRun()
}
```

It is possible to use an alternative font by specifying a `FYNE_FONT` environment variable. Use this to set a `.ttf` file to use instead of the one provided in the Fyne toolkit or the current theme.

Line

The `canvas.Line` object draws a line from the `Position1` (default is top, left) to `Position2` (default is bottom, right). You specify its colour and can vary the stroke width which otherwise defaults to `1`.

A line position can be manipulated using the `Position1` or `Position2` fields or by using the `Move()` and `Resize()` functions. For example a 0 width area will show a vertical line whereas 0 height would be horizontal.

```
package main

import (
        "image/color"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
)

func main() {
        myApp := app.New()
        w := myApp.NewWindow("Line")

        line := canvas.NewLine(color.White)
        line.StrokeWidth = 5
        w.SetContent(line)

        w.Resize(fyne.NewSize(100, 100))
        w.ShowAndRun()
}
```

Lines are typically used in a custom layout or controlled manually. Unlike text they have no natural (minimum) size but can be used to great effect in complex layouts.

Circle

`canvas.Circle` defines a circle shape filled by the specified colour. You may also set a `StrokeWidth` and therefore a different `StrokeColor` as shown in this example.

The circle will fill the space specified by calling `Resize()` or by the layout it is controlled by. As the example sets the circle as the window content it will resize to fill the window, within a basic padding (controlled by the theme).

```
package main

import (
        "image/color"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
)

func main() {
        myApp := app.New()
        w := myApp.NewWindow("Circle")

        circle := canvas.NewCircle(color.White)
        circle.StrokeColor = color.Gray{Y: 0x99}
        circle.StrokeWidth = 5
        w.SetContent(circle)

        w.Resize(fyne.NewSize(100, 100))
        w.ShowAndRun()
}
```

All these have been basic types that can be rendered by our driver with no additional information. Next we will look at more complex types starting with `Image`.

Image

A `canvas.Image` represents a scalable image resource in Fyne. It can be loaded from a resource (as shown in the example), from an image file, from a URI location containing an image, from an `io.Reader`, or from a Go `image.Image` in memory.

The default image fill mode is `canvas.ImageFillStretch` which will cause it to fill the space specified (through `Resize()` or layout). Alternatively you could use `canvas.ImageFillContain` to ensure that the aspect ratio is maintained and the image is within the bounds. Further to this you can use `canvas.ImageFillOriginal` (as used in the example here) which ensures that it will also have a minimum size equal to that of the original image size.

```
package main

import (
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
        "fyne.io/fyne/v2/theme"
)

func main() {
        myApp := app.New()
        w := myApp.NewWindow("Image")

        image := canvas.NewImageFromResource(theme.FyneLogo())
        // image := canvas.NewImageFromURI(uri)
        // image := canvas.NewImageFromImage(src)
        // image := canvas.NewImageFromReader(reader, name)
        // image := canvas.NewImageFromFile(fileName)
        image.FillMode = canvas.ImageFillOriginal
        w.SetContent(image)

        w.ShowAndRun()
}
```

Images can be bitmap based (like PNG and JPEG) or vector based (such as SVG). Where possible we recommend scalable images as they will continue to render well as sizes change. Be careful when using original image sizes as they may not behave exactly as expected with different user interface scales. As Fyne allows the entire user interface to scale, a 25px image file may not be the same height as a 25 height fyne object.

Raster

The `canvas.Raster` is like an image but draws exactly one spot for each pixel on the screen. This means that as a user interface scales or the image resizes more pixels will be requested to fill the space. To do this we use a `Generator` function as illustrated in this example - it will be used to return the colour of each pixel.

The generator functions can be pixel based (as in this example where we generate a new random colour for each pixel) or based on full images. Generating complete images (with `canvas.NewRaster()`) is more efficient but sometimes controlling pixels directly is more convenient.

```
package main

import (
        "image/color"
        "math/rand"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
)

func main() {
        myApp := app.New()
        w := myApp.NewWindow("Raster")

        raster := canvas.NewRasterWithPixels(
                func(_, _, w, h int) color.Color {
                        return color.RGBA{R: uint8(rand.Intn(255)),
                                G: uint8(rand.Intn(255)),
                                B: uint8(rand.Intn(255)), A: 0xff}
```

```
                })
        // raster := canvas.NewRasterFromImage()
        w.SetContent(raster)
        w.Resize(fyne.NewSize(120, 100))
        w.ShowAndRun()
}
```

If your pixel data is stored in an image you can load it through the `NewRasterFromImage()` function which will load the image to display pixel perfect on screen.

Gradient

The last canvas primitive type is Gradient, available as `canvas.LinearGradient` and `canvas.RadialGradient` which is used to draw a gradient from one colour to another in various patterns. You can create gradients using `NewHorizontalGradient()`, `NewVerticalGradient()` or `NewRadialGradient()`.

To create a gradient you need a start and end colour - every colour in between is calculated by the canvas. In this example we use `color.Transparent` to show how a gradient (or any other type) could use an alpha value to be semi-transparent over the content behind.

```
package main

import (
        "image/color"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
)

func main() {
        myApp := app.New()
        w := myApp.NewWindow("Gradient")

        gradient := canvas.NewHorizontalGradient(color.White, color.Transparent)
        //gradient := canvas.NewRadialGradient(color.White, color.Transparent)
        w.SetContent(gradient)

        w.Resize(fyne.NewSize(100, 100))
        w.ShowAndRun()
}
```

Animation

Fyne includes an animation framework that allows you to smoothly transition canvas properties from one value to another over time. An animation can contain any code which means that any types of object attributes can be managed, however there are builtin animations for size, position and color.

Animations are normally created using the builtin helpers of the canvas package, such as `NewSizeAnimation`, and calling `Start()` on the created animation. You can set animations to repeat or auto reverse, as we will see below.

Let us look first at a colour animation which gradually changes the fill colour of a `Rectangle`. In the following code sample we set an rectangle to be set as the content of a window, as we have done in earlier code samples. The big difference is the animation that we start just before showing the window. The animation is created using `NewColorRGBAAnimation` which will transition the colour channels from the defined `red` state through to `blue` and it will take 2 seconds (the specified duration) to do so.

```
package main

import (
        "image/color"
        "time"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
        "fyne.io/fyne/v2/container"
)
```

```
func main() {
	a := app.New()
	w := a.NewWindow("Hello")

	obj := canvas.NewRectangle(color.Black)
	obj.Resize(fyne.NewSize(50, 50))
	w.SetContent(container.NewWithoutLayout(obj))

	red := color.NRGBA{R:0xff, A:0xff}
	blue := color.NRGBA{B:0xff, A:0xff}
	canvas.NewColorRGBAAnimation(red, blue, time.Second*2, func(c color.Color) {
		obj.FillColor = c
		canvas.Refresh(obj)
	}).Start()

	w.Resize(fyne.NewSize(250, 50))
	w.SetPadded(false)
	w.ShowAndRun()
}
```

It is also possible to animate multiple properties at the same time. If you look carefully you will see that we added the rectangle into a container without layout - this means that we can manually move or resize the object. Let's add a new position animation that will move the `Rectangle` across the window, and automatically reverse as well.

```
move := canvas.NewPositionAnimation(fyne.NewPos(0, 0), fyne.NewPos(200, 0), time.Second,
obj.Move)
move.AutoReverse = true
move.Start()
```

Because the `Move()` function of `CanvasObject` expects a `fyne.Position` argument, and so does the position animation callback, we can simply pass the method name instead of creating a new function If you add the code above just under the first animation you will see that the object moves across the window at the same time as it changes colour!

Box

As discussed in Container and Layouts elements within a container can be arranged using a layout. This section explores the builtin layouts and how to use them.

The most commonly used layout is `layout.BoxLayout` and it has two variants, horizontal and vertical. A box layout arranges all elements in a single row or column with optional spaces to assist alignment.

A horizontal box layout, created with `layout.NewHBoxLayout()` creates an arrangement of items in a single row. Every item in the box will have its width set to its `MinSize().Width` and the height will be equal for all items, the largest of all the `MinSize().Height` values. The layout can be used in a container or you can use the box widget `widget.NewHBox()`.

A vertical box layout is similar but it arranges items in a column. Each item will have its height set to minimum and all the widths will be equal, set to the largest of the minimum widths.

To create an expanding space between elements (so that some are left aligned and the others right aligned, for example) add a `layout.NewSpacer()` as one of the items. A spacer will expand to fill all available space. Adding a spacer at the beginning of a vertical box layout will cause all items to be bottom aligned. You can add one to the beginning and end of a horizontal arrangement to create a center alignment.

```
package main

import (
	"image/color"

	"fyne.io/fyne/v2/app"
	"fyne.io/fyne/v2/canvas"
	"fyne.io/fyne/v2/container"
	"fyne.io/fyne/v2/layout"
)

func main() {
	myApp := app.New()
	myWindow := myApp.NewWindow("Box Layout")
```

```
        text1 := canvas.NewText("Hello", color.White)
        text2 := canvas.NewText("There", color.White)
        text3 := canvas.NewText("(right)", color.White)
        content := container.New(layout.NewHBoxLayout(), text1, text2,
layout.NewSpacer(), text3)

        text4 := canvas.NewText("centered", color.White)
        centered := container.New(layout.NewHBoxLayout(), layout.NewSpacer(), text4,
layout.NewSpacer())
        myWindow.SetContent(container.New(layout.NewVBoxLayout(), content, centered))
        myWindow.ShowAndRun()
}
```

Grid

The grid layout lays out the elements of a container in a grid pattern with a fixed number of columns. Items will fill a single row until the number of columns is met, after this a new row will be created. Vertical space will be split equally between each of the rows of objects.

You create a grid layout using `layout.NewGridLayout(cols)` where cols is the number of items (columns) you wish to have in each row. This layout is then passed as the first parameter to `container.New(...)`.

If you resize the container then each of the cells will resize equally to share the available space.

```
package main

import (
        "image/color"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/layout"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Grid Layout")

        text1 := canvas.NewText("1", color.White)
        text2 := canvas.NewText("2", color.White)
        text3 := canvas.NewText("3", color.White)
        grid := container.New(layout.NewGridLayout(2), text1, text2, text3)
        myWindow.SetContent(grid)
        myWindow.ShowAndRun()
}
```

Grid Wrap

Like the previous grid layout, the grid wrap layout creates an arrangement of elements in a grid pattern. However this grid does not have a set number of columns, instead it uses a fixed size for each cell and then flows the content to as many rows as is needed to display the items.

You create a grid wrap layout using `layout.NewGridWrapLayout(size)` where size specifies the size to apply to all child elements. This layout is then passed as the first parameter to `container.New(...)`. The number of columns and rows will be calculated based on the current size of the container.

Initially a grid wrap layout will have a single column, if you resize it (as illustrated in the code comment to the right) it will rearrange the child elements to fill the space.

```
package main

import (
        "image/color"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
```

```
        "fyne.io/fyne/v2/canvas"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/layout"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Grid Wrap Layout")

        text1 := canvas.NewText("1", color.White)
        text2 := canvas.NewText("2", color.White)
        text3 := canvas.NewText("3", color.White)
        grid := container.New(layout.NewGridWrapLayout(fyne.NewSize(50, 50)),
                text1, text2, text3)
        myWindow.SetContent(grid)

        // myWindow.Resize(fyne.NewSize(180, 75))
        myWindow.ShowAndRun()
}
```

Border

The border layout is possibly the most widely used to construct user interfaces as it allows the positioning of items around a central element which will expand to fill the space. To create a border container you need to pass the `fyne.CanvasObject`s that should be positioned in a border position to the constructor's first four parameters. This syntax is basically just `container.NewBorder(top, bottom, left, right, center)` as illustrated in the example.

Any items passed to the container after the first four items will be positioned to the central area and will expand to fill the space available. You can also pass `nil` to border parameters that you wish to leave empty.

```
package main

import (
        "image/color"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/layout"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Border Layout")

        top := canvas.NewText("top bar", color.White)
        left := canvas.NewText("left", color.White)
        middle := canvas.NewText("content", color.White)
        content := container.NewBorder(top, nil, left, nil, middle)
        myWindow.SetContent(content)
        myWindow.ShowAndRun()
}
```

Note that all items in the center will expand to fill the space (as if they were in a `layout.MaxLayout` container). To manage the area yourself you can use any `fyne.Container` as the content instead.

Form

The `layout.FormLayout` is like a 2 column grid layout but tweaked to lay out forms in an application. The height of each item will be the larger of the two minimum heights in each row. The width of the left item will be the largest minimum width of all items in the first column whilst the second item in each row will expand to fill the space.

This layout is more typically used within the `widget.Form` (for validation, submit and cancel buttons, etc) but it can also be used directly with `layout.NewFormLayout()` passed to the first parameter of `container.New(...)`.

```
package main
```

```
import (
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/layout"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Form Layout")

        label1 := widget.NewLabel("Label 1")
        value1 := widget.NewLabel("Value")
        label2 := widget.NewLabel("Label 2")
        value2 := widget.NewLabel("Something")
        grid := container.New(layout.NewFormLayout(), label1, value1, label2, value2)
        myWindow.SetContent(grid)
        myWindow.ShowAndRun()
}
```

Center

`layout.CenterLayout` organises all items in its container to be centered in the available space. The objects will be drawn in the order they are passed to the container, with the last being drawn top-most.

```
package main

import (
        "image/color"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/layout"
        "fyne.io/fyne/v2/theme"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Center Layout")

        img := canvas.NewImageFromResource(theme.FyneLogo())
        img.FillMode = canvas.ImageFillOriginal
        text := canvas.NewText("Overlay", color.Black)
        content := container.New(layout.NewCenterLayout(), img, text)

        myWindow.SetContent(content)
        myWindow.ShowAndRun()
}
```

The center layout causes all items to stay at their minimum size, if you wish to expand items to fill the space then see `layout.MaxLayout`.

Stack

The `layout.NewStackLayout()` is the simplest layout, it sets all items in the container to be the same size as the container. This is not often useful in general containers but can be suitable when composing widgets.

The stack layout will require the container to be at least the size of the largest item's minimum size. The objects will be drawn in the order the are passed to the container, with the last being drawn top-most.

```
package main

import (
        "image/color"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/canvas"
```

```
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/layout"
        "fyne.io/fyne/v2/theme"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Stack Layout")

        img := canvas.NewImageFromResource(theme.FyneLogo())
        text := canvas.NewText("Overlay", color.Black)
        content := container.New(layout.NewStackLayout(), img, text)

        myWindow.SetContent(content)
        myWindow.ShowAndRun()
}
```

AppTabs

The AppTabs container is used to allow the user to switch between different content panels. Tabs are either just text or text and an icon. It is recommended not to mix some tabs having icons and some without. A tab container is created using `container.NewAppTabs(...)` and passing `container.TabItem` items (that can be created using `container.NewTabItem(...)`).

The tab container can be configured by setting the location of tabs, one of `container.TabLocationTop`, `container.TabLocationBottom`, `container.TabLocationLeading` and `container.TabLocationTrailing`. The default location is top.

```
package main

import (
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/container"
        //"fyne.io/fyne/v2/theme"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("TabContainer Widget")

        tabs := container.NewAppTabs(
                container.NewTabItem("Tab 1", widget.NewLabel("Hello")),
                container.NewTabItem("Tab 2", widget.NewLabel("World!")),
        )

        //tabs.Append(container.NewTabItemWithIcon("Home", theme.HomeIcon(),
widget.NewLabel("Home tab")))

        tabs.SetTabLocation(container.TabLocationLeading)

        myWindow.SetContent(tabs)
        myWindow.ShowAndRun()
}
```

When loaded on a mobile device the tab location may be ignored. In a portrait orientation a leading or trailing position will be changed to bottom. When in landscape orientation, the top or bottom positions will be moved to leading.

Label

Widgets are the main components of a Fyne application GUI, they can be used in any place that a basic `fyne.CanvasObject` can. They manage user interactions and will always match the current theme.

The Label widget is the simplest of them - it presents text to the user. Unlike `canvas.Text` it can handle some simple formatting (such as `\n`) and wrapping (by setting the `Wrapping` field). You can create a label by calling `widget.NewLabel("some text")`, the result can be assigned to a variable or passed directly into a container.

```
package main

import (
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Label Widget")

        content := widget.NewLabel("text")

        myWindow.SetContent(content)
        myWindow.ShowAndRun()
}
```

Button

The button widget can contain text, an icon or both, the constructor functions are `widget.NewButton()` and `widget.NewButtonWithIcon()`. To create a text button there are just 2 parameters, the `string` content and a 0 parameter `func()` that will be called when the button is tapped. See the example for how that can be created.

The button constructor with an icon includes an additional parameter which is the `fyne.Resource` which contains the icon data. The builtin icons within the `theme` package all adapt appropriately to a change in theme. You can pass in your own image if it is loaded as a resource - helpers such as `fyne.LoadResourceFromPath()` may assist, though bundling resources is recommended where possible.

To create a button with only an icon you should pass “” as the label parameter to `widget.NewButtonWithIcon()`.

```
package main

import (
        "log"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/widget"
        //"fyne.io/fyne/v2/theme"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Button Widget")

        content := widget.NewButton("click me", func() {
                log.Println("tapped")
        })

        //content := widget.NewButtonWithIcon("Home", theme.HomeIcon(), func() {
        //      log.Println("tapped home")
        //})

        myWindow.SetContent(content)
        myWindow.ShowAndRun()
}
```

Entry

The entry widget is used for user input of simple text content. An entry can be created with a simple `widget.NewEntry()` constructing function. When you create the widget keep a reference so that you can access its `Text` field later. It is also possible to use the `OnChanged` callback function to be notified every time the content changes.

Entry widgets can also have validation for verifying the text input typed into it. This can be done by setting the `Validator` field to a `fyne.StringValidator`. You can also set a `PlaceHolder` text and also set the entry to `MultiLine` to accept more than one line of text.

```
package main
```

```
import (
        "log"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Entry Widget")

        input := widget.NewEntry()
        input.SetPlaceHolder("Enter text...")

        content := container.NewVBox(input, widget.NewButton("Save", func() {
                log.Println("Content was:", input.Text)
        }))

        myWindow.SetContent(content)
        myWindow.ShowAndRun()
}
```

You can also create a password entry (where the content is obscured) using the `NewPasswordEntry()` function.

Choices

There are various widgets available to present the user with a choice, these include a checkbox, radio group and select popup.

The `widget.Check` provides a simple yes/no choice and is created using a string label. Each of these widgets also takes a "changed" `func(...)` where the parameter is of the appropriate type. `widget.NewCheck(..)` therefore takes a `string` parameter for the label and a `func(bool)` param for the change handler. You can also use the `Checked` field to get the boolean value.

The radio widget is similar, but the first parameter is a slice of `string`s that represents each of the options. The change function expects a `string` parameter this time to return the currently selected value. Call `widget.NewRadioGroup(...)` to construct the radio group widget, you can use this reference later to read the `Selected` field instead of using the change callback.

The select widget is identical in the constructor signature as the radio widget. Calling `widget.NewSelect(...)` will instead show a button that displays a popup when tapped from which the user can make a selection. This is better for long lists of options.

```
package main

import (
        "log"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Choice Widgets")

        check := widget.NewCheck("Optional", func(value bool) {
                log.Println("Check set to", value)
        })
        radio := widget.NewRadioGroup([]string{"Option 1", "Option 2"}, func(value
string) {
                log.Println("Radio set to", value)
        })
        combo := widget.NewSelect([]string{"Option 1", "Option 2"}, func(value string) {
                log.Println("Select set to", value)
        })

        myWindow.SetContent(container.NewVBox(check, radio, combo))
        myWindow.ShowAndRun()
```

```
}
```

Form

The form widget is used to lay out many input fields with labels and optional cancel and submit buttons. In its most bare form it aligns labels to the left of each input widget. By setting OnCancel or OnSubmit the form will add a button bar with the specified handlers called when appropriate.

The widget can be created with `widget.NewForm(...)` passing a list of `widget.FormItem`s, or by using the `&widget.Form{}` syntax illustrated in the example. There is also a helpful `Form.Append(label, widget)` that can be used for an alternative syntax.

In this example we create two entries, one of which is a “multiline” (like HTML TextArea) to hold values. There is an OnSubmit handler which prints the information before closing the window (and therefore the application).

```
package main

import (
        "log"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Form Widget")

        entry := widget.NewEntry()
        textArea := widget.NewMultiLineEntry()

        form := &widget.Form{
                Items: []*widget.FormItem{ // we can specify items in the constructor
                        {Text: "Entry", Widget: entry}},
                OnSubmit: func() { // optional, handle form submission
                        log.Println("Form submitted:", entry.Text)
                        log.Println("multiline:", textArea.Text)
                        myWindow.Close()
                },
        }

        // we can also append items
        form.Append("Text", textArea)

        myWindow.SetContent(form)
        myWindow.ShowAndRun()
}
```

ProgressBar

The progress bar widget has two forms, the standard progress bar shows the user which `Value` has been reached, from `Min` to `Max`. The default min is `0.0` and the max defaults to `1.0`. To use the default values just call `widget.NewProgressBar()`. After creating you can set the `Value` field.

To set up a custom range you can set `Min` and `Max` fields manually. The label will always show the percentage completion.

The other form of progress widget is the infinite progress bar. This version simply shows that some activity is ongoing by moving a segment of the bar from left to right and back again. You create this using `widget.NewProgressBarInfinite()` and it will start animating as soon as it is shown.

```
package main

import (
        "time"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/container"
```

```
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("ProgressBar Widget")

        progress := widget.NewProgressBar()
        infinite := widget.NewProgressBarInfinite()

        go func() {
                for i := 0.0; i <= 1.0; i += 0.1 {
                        time.Sleep(time.Millisecond * 250)
                        progress.SetValue(i)
                }
        }()

        myWindow.SetContent(container.NewVBox(progress, infinite))
        myWindow.ShowAndRun()
}
```

Toolbar

The toolbar widget creates a row of action buttons using icons to represent each. The `widget.NewToolbar(...)` constructor function takes a list of `widget.ToolbarItem` parameters. The builtin types of toolbar items are action, separator and spacer.

The most used item is an action that is created using the `widget.NewToolbarItemAction(..)` function. An action takes two parameters, first being the icon resource to draw and the latter is the `func()` to call when tapped. This creates a standard toolbar button.

You can use `widget.NewToolbarSeparator()` to create a small divider between items in a toolbar (usually a thin vertical line). Lastly you can use `widget.NewToolbarSpacer()` to create a flexible space between elements. This is most useful to right align the toolbar items that are listed after the spacer.

A toolbar should always be at the top of the content area so it's normal to add it to a `fyne.Container` using the `layout.BorderLayout` to align it above other content.

```
package main

import (
        "log"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/theme"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Toolbar Widget")

        toolbar := widget.NewToolbar(
                widget.NewToolbarAction(theme.DocumentCreateIcon(), func() {
                        log.Println("New document")
                }),
                widget.NewToolbarSeparator(),
                widget.NewToolbarAction(theme.ContentCutIcon(), func() {}),
                widget.NewToolbarAction(theme.ContentCopyIcon(), func() {}),
                widget.NewToolbarAction(theme.ContentPasteIcon(), func() {}),
                widget.NewToolbarSpacer(),
                widget.NewToolbarAction(theme.HelpIcon(), func() {
                        log.Println("Display help")
                }),
        )

        content := container.NewBorder(toolbar, nil, nil, nil,
widget.NewLabel("Content"))
        myWindow.SetContent(content)
```

```
        myWindow.ShowAndRun()
}
```

List

The `List` collection widget is one of the toolkit's collection widgets. These widgets are designed to help build really performant interfaces when lots of data is being presented. You can also see the Table and Tree widgets which have a similar API. Because of this design they are a little more complicated to use.

The `List` uses callback functions to ask for data when it is required. There are 3 main callbacks, `Length`, `CreateItem` and `UpdateItem`. The Length callback (passed first) is the simplest, it returns how many items are in the data to be presented. The others relate to templates - how graphical elements are created, cached and re-used.

The `CreateItem` callback returns a new template object. This will be re-used with real data when the widget is presented. The `MinSize` of this object will influence the `List` minimum size. Lastly `UpdateItem` is called to apply an item of data to a cached template. Use this to set the content ready for display.

```
package main

import (
        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/widget"
)

var data = []string{"a", "string", "list"}

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("List Widget")

        list := widget.NewList(
                func() int {
                        return len(data)
                },
                func() fyne.CanvasObject {
                        return widget.NewLabel("template")
                },
                func(i widget.ListItemID, o fyne.CanvasObject) {
                        o.(*widget.Label).SetText(data[i])
                })

        myWindow.SetContent(list)
        myWindow.ShowAndRun()
}
```

Table

The `Table` collection widget is like the List widget (another of the toolkit's collection widgets) with a two-dimensional index. Like `List` this is designed to help build really performant interfaces when lots of data is being presented. Because of this the widget is not created with all the data embedded, but instead calls out to the data source when needed.

The `Table` uses callback functions to ask for data when it is required. There are 3 main callbacks, `Length`, `CreateCell` and `UpdateCell`. The Length callback (passed first) is the simplest, it returns how many items are in the data to be presented, the two ints it returns represent the row and column count. The other two relate to the content templates.

The `CreateCell` callback returns a new template object, just like list. The difference being that `MinSize` will define the standard size of each cell, and the minimum size of the table (it shows at least one cell). As previously the `UpdateCell` is called to apply data to a cell template. The index passed in is the same `(row, col)` int pair.

```
package main

import (
        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/widget"
)
```

```
var data = [][]string{[]string{"top left", "top right"},
        []string{"bottom left", "bottom right"}}

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Table Widget")

        list := widget.NewTable(
                func() (int, int) {
                        return len(data), len(data[0])
                },
                func() fyne.CanvasObject {
                        return widget.NewLabel("wide content")
                },
                func(i widget.TableCellID, o fyne.CanvasObject) {
                        o.(*widget.Label).SetText(data[i.Row][i.Col])
                })

        myWindow.SetContent(list)
        myWindow.ShowAndRun()
}
```

For more info, for example on how to add headers to the table, see the widget.Table API documentation.

Tree

The `Tree` collection widget is like the List widget (another of the toolkit's collection widgets) with a multi-level data structure. Like `List` this is designed to help build really performant interfaces when lots of data is being presented. Because of this the widget is not created with all the data embedded, but instead calls out to the data source when needed.

The `Tree` uses callback functions to ask for data when it is required. There are 4 main callbacks, `ChildUIDs`, `IsBranch`, `CreateNode` and `UpdateNode`. The ChildUIDs callback is where we communicate the unique ID of each child node to the one requested. This will be called with the `TreeNodeID` `""` to first get a list of all the IDs that appear inside the root of the tree. The `IsBranch` callback should return `true` if the node is a branch. This is usually true if the node ID has child nodes - but you can have an empty branch.

It is crucial that the id of each tree node is unique to the tree. For example if you are building a file manager the ID should be the file path rather than its name.

The other two relate to the content templates.

The `CreateNode` callback returns a new template object, just like list, though there is an additional `bool` parameter that is `true` if the node can have child elements (is a branch). As previously the `UpdateNode` is called to apply data to a cell template. You should update the content according to the `TreeNodeID` and `isBranch` parameters.

```
package main

import (
        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        myWindow := myApp.NewWindow("Table Widget")

        tree := widget.NewTree(
                func(id widget.TreeNodeID) []widget.TreeNodeID {
                        switch id {
                        case "":
                                return []widget.TreeNodeID{"a", "b", "c"}
                        case "a":
                                return []widget.TreeNodeID{"a1", "a2"}
                        }
                        return []string{}
                },
```

```
                func(id widget.TreeNodeID) bool {
                        return id == "" || id == "a"
                },
                func(branch bool) fyne.CanvasObject {
                        if branch {
                                return widget.NewLabel("Branch template")
                        }
                        return widget.NewLabel("Leaf template")
                },
                func(id widget.TreeNodeID, branch bool, o fyne.CanvasObject) {
                        text := id
                        if branch {
                                text += " (branch)"
                        }
                        o.(*widget.Label).SetText(text)
                })

        myWindow.SetContent(tree)
        myWindow.ShowAndRun()
}
```

Data Binding

Data binding is a powerful new addition to the Fyne toolkit that was introduced in version `v2.0.0`. By using data binding we can avoid manually managing many standard objects like `Label`s, `Button`s and `List`s.

The builtin bindings support many primitive types (like `Int`, `String`, `Float` etc), lists (such as `StringList`, `BoolList`) as well as `Map` and `Struct` bindings. Each of these types can be created using a simple constructor function. For example to create a new string binding with a zero value you can use `binding.NewString()`. You can get or set the value of most data bindings using `Get` and `Set` methods.

It is also possible to bind to an existing value using similar functions who's names start `Bind` and they all accept a pointer to the type bound. To bind to an existing `int` we could use `binding.BindInt(&myInt)`. By keeping a reference to a bound value instead of the original variable we can configure widgets and functions to respond to any changes automatically. If you change the external data directly, be sure to call `Reload`() to ensure that the binding system reads the new value.

```
package main

import (
        "log"

        "fyne.io/fyne/v2/data/binding"
)

func main() {
        boundString := binding.NewString()
        s, _ := boundString.Get()
        log.Printf("Bound = '%s'", s)

        myInt := 5
        boundInt := binding.BindInt(&myInt)
        i, _ := boundInt.Get()
        log.Printf("Source = %d, bound = %d", myInt, i)
}
```

Binding Simple Widgets

The simplest form of binding a widget is to pass it a bound item as a value instead of a static value. Many widgets provide a `WithData` constructor that will accept a typed data binding item. To set up the binding all you need to do is pass the right type in.

Although this may not look like much of a benefit in the initial code you can then see how it ensures that the displayed content is always up to date with the source of the data. You will notice how we did not need to call `Refresh()` on the `Label` widget or even keep a reference to it and yet it updates appropriately.

```
package main

import (
```

```
        "time"

        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/data/binding"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        w := myApp.NewWindow("Simple")

        str := binding.NewString()
        str.Set("Initial value")

        text := widget.NewLabelWithData(str)
        w.SetContent(text)

        go func() {
                time.Sleep(time.Second * 2)
                str.Set("A new string")
        }()

        w.ShowAndRun()
}
```

Two-Way Binding

So far we have looked at data binding as a way of keeping our user interface elements up to date. Far more common, however, is the need to update values from the UI widgets and keep the data up to date everywhere. Thankfully the bindings provided in Fyne are “two-way” which means that values can be pushed into them as well as read out. The change in data will be communicated to all connected code without any additional code.

To see this in action we can update the last test app to display a `Label` and an `Entry` that are bound to the same value. By setting this up you can see that editing the value through the entry will update the text in the label as well. This is all possible without calling refresh or referencing the widgets in our code.

By moving your app to use data binding you can stop saving pointers to all your widgets. By instead capturing your data as a set of bound values your user interface can be completely separate code. Cleaner to read and easier to manage.

```
package main

import (
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/data/binding"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        w := myApp.NewWindow("Two Way")

        str := binding.NewString()
        str.Set("Hi!")

        w.SetContent(container.NewVBox(
                widget.NewLabelWithData(str),
                widget.NewEntryWithData(str),
        ))

        w.ShowAndRun()
}
```

Next we will look at how to add conversions in our data.

Data Conversion

So far we have used data binding where the type of data matches the output type (for example `String` and `Label` or `Entry`). Often it will be desirable to present data that is not already in the correct format. To do this the `binding` package provides a number of helpful conversion functions.

Most commonly this will be used to convert different types of data into strings for displaying in `Label` or `Entry` widgets. See in the code how we can convert a `Float` to `String` using `binding.FloatToString`. The original value can be edited by moving the slider. Each time the data changes it will run the conversion code and update any connected widgets.

It is also possible to use format strings to add more natural output for the user interface. You can see that our `short` binding is also converting a `Float` to `String` but by using a `WithFormat` helper we can pass a format string (similar to the `fmt` package) to provide a custom output.

```
package main

import (
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/data/binding"
        "fyne.io/fyne/v2/widget"
)

func main() {
        myApp := app.New()
        w := myApp.NewWindow("Conversion")

        f := binding.NewFloat()
        str := binding.FloatToString(f)
        short := binding.FloatToStringWithFormat(f, "%0.0f%%")
        f.Set(25.0)

        w.SetContent(container.NewVBox(
                widget.NewSliderWithData(0, 100.0, f),
                widget.NewLabelWithData(str),
                widget.NewLabelWithData(short),
        ))

        w.ShowAndRun()
}
```

Lastly in this section we will look at list data.

List Data

To demonstrate how more complex types can be connected we will look at the `List` widget and how data binding can make it easier to use. Firstly we create a `StringList` data binding, which is a list of `String` data type. Once we have a data of list type we can connect this to the standard `List` widget. To do so we use the `widget.NewListWithData` constructor, much like other widgets.

Comparing this code to the list tour You will see 2 main changes, the first is that we pass the data type as the first parameter instead of a length callback function. The second change is the last parameter, our `UpdateItem` callback. The revised version takes a `binding.DataItem` value instead of `widget.ListIndexID`. When using this callback structure we should `Bind` to the template label widget instead of calling `SetText`. This means that if any of the strings change in the data source each affected row of the table will refresh.

```
package main

import (
        "fmt"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/data/binding"
        "fyne.io/fyne/v2/widget"
)

func main() {
```

```
        myApp := app.New()
        myWindow := myApp.NewWindow("List Data")

        data := binding.BindStringList(
                &[]string{"Item 1", "Item 2", "Item 3"},
        )

        list := widget.NewListWithData(data,
                func() fyne.CanvasObject {
                        return widget.NewLabel("template")
                },
                func(i binding.DataItem, o fyne.CanvasObject) {
                        o.(*widget.Label).Bind(i.(binding.String))
                })

        add := widget.NewButton("Append", func() {
                val := fmt.Sprintf("Item %d", data.Length()+1)
                data.Append(val)
        })
        myWindow.SetContent(container.NewBorder(nil, add, nil, nil, list))
        myWindow.ShowAndRun()
}
```

In our demo code there is an “Append” `Button`, when tapped it will append a new value to the data source. Doing so will automatically trigger the data change handlers and expand the `List` widget to display the new data.

Building a Custom Layout

In a Fyne application each `Container` arranges its child elements using a simple layout algorithm. Fyne defines many layouts available in the `fyne.io/fyne/v2/layout` package. If you look at the code you will see that they all implement the `Layout` interface.

```
type Layout interface {
        Layout([]CanvasObject, Size)
        MinSize(objects []CanvasObject) Size
}
```

Any application can provide a custom layout to arrange widgets in a non-standard manner. To do this you need to implement the interface above in your own code. To illustrate this we will create a new layout that arranges elements in a diagonal line and is arranged to the bottom left of its container

First we will define a new type, `diagonal`, and define what its minimum size will be. To calculate this we just add the width and height of all child elements (specified as the `[]fyne.CanvasObject` parameter to `MinSize`).

```
import "fyne.io/fyne/v2"

type diagonal struct {
}

func (d *diagonal) MinSize(objects []fyne.CanvasObject) fyne.Size {
        w, h := float32(0), float32(0)
        for _, o := range objects {
                childSize := o.MinSize()

                w += childSize.Width
                h += childSize.Height
        }
        return fyne.NewSize(w, h)
}
```

To this type we add a `Layout()` function that should move all of the specified objects into the `fyne.Size` specified in the second parameter.

In our implementation we calculate the top-left of the widgets (this is `0` x parameter and has a y position that is the height of the container less the total of all child item heights). From the top position we simply advance each item position by the size of the previous child.

```
func (d *diagonal) Layout(objects []fyne.CanvasObject, containerSize fyne.Size) {
```

```
        pos := fyne.NewPos(0, containerSize.Height - d.MinSize(objects).Height)
        for _, o := range objects {
                size := o.MinSize()
                o.Resize(size)
                o.Move(pos)

                pos = pos.Add(fyne.NewPos(size.Width, size.Height))
        }
}
```

That's all there is to creating a custom layout. Now that the code is all written we can use it as the `layout` parameter to `container.New`. The code below sets up 3 `Label` widgets and place them in a container with our new layout. If you run this application you will see the diagonal widget arrangement and, upon resizing the window, they will align to the bottom left of the available space.

```
package main

import (
        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/container"
        "fyne.io/fyne/v2/widget"
)

func main() {
        a := app.New()
        w := a.NewWindow("Diagonal")

        text1 := widget.NewLabel("topleft")
        text2 := widget.NewLabel("Middle Label")
        text3 := widget.NewLabel("bottomright")

        w.SetContent(container.New(&diagonal{}, text1, text2, text3))
        w.ShowAndRun()
}
```

Writing a Custom Widget

The standard widgets included with Fyne are designed to support standard user interactions and requirements. As a GUI often has to provide custom functionality it may be necessary to write a custom widget. This article outlines how.

A widget is split into two areas - each implementing a standard interface - the `fyne.Widget` and the `fyne.WidgetRenderer`. The widget defines behaviour and state, with the renderer being used to define how it should be drawn to screen.

fyne.Widget

A widget in Fyne is simply a stateful canvas object that has its rendering definition separated from the main logic. As you can see from the `fyne.Widget` interface there is not much that must be implemented.

```
type Widget interface {
        CanvasObject

        CreateRenderer() WidgetRenderer
}
```

As a widget needs to be used like any other canvas item we inherit from the same interface. To save writing all the functions required we can make use of the `widget.BaseWidget` type which handles the basics.

Each widget definition will contain much more than the interface requires. It is standard in a Fyne widget to export the fields which define behaviour (just like the primitives defined in the `canvas` package).

For example, look at the `widget.Button` type:

```
type Button struct {
        BaseWidget
        Text  string
        Style ButtonStyle
        Icon  fyne.Resource

        OnTapped func()
}
```

You can see how each of these items store state about the widget behaviour but nothing about how it is rendered.

fyne.WidgetRenderer

The widget renderer is responsible for managing a list of `fyne.CanvasObject` primitives that come together to create the design of our widget. It is much like a `fyne.Container` with a custom layout and some additional theme handling.

Every widget must provide a renderer, but it is entirely possible to re-use a renderer from another widget - especially if your widget is a lightweight wrapper around another standard control.

```
type WidgetRenderer interface {
        Layout(Size)
        MinSize() Size

        Refresh()
        Objects() []CanvasObject
        Destroy()
}
```

As you can see the `Layout(Size)` and `MinSize()` functions are similar to the `fyne.Layout` interface, without the `[]fyne.CanvasObject` parameter - this is because a widget does need to be laid out but it controls which objects will be included.

The `Refresh()` method is triggered when the widget this renderer draws has changed or if the theme is altered. In either situation we may need to make adjustments to how it looks. Lastly the `Destroy()` method is called when this renderer is no longer needed so it should clear any resources that would otherwise leak.

Compare again with the button widget - it's `fyne.WidgetRenderer` implementation is based on the following type:

```
type buttonRenderer struct {
        icon   *canvas.Image
        label  *canvas.Text
```

```
        shadow *fyne.CanvasObject

        objects []fyne.CanvasObject
        button  *Button
}
```

As you can see it has fields to cache the actual image, text and shadow canvas objects for drawing. It keeps track of the slice of objects required by `fyne.WidgetRenderer` as a convenience.

Lastly it keeps a reference to the `widget.Button` for all state information. In the `Refresh()` method it will update the graphical state based on any changes in the underlying `widget.Button` type.

Bring it together

A basic widget will extend the `widget.BaseWidget` type and declare any state that the widget holds. The `CreateRenderer()` function must exist and return a new `fyne.WidgetRenderer` instance. The widget and driver code in Fyne will ensure that this is cached accordingly - this method may be called many times (for example if a widget is hidden and then shown). If `CreateRenderer()` is called again you should return a new renderer instance as the old one may have been destroyed.

Take care not to keep any important state in your renderer - animation tickers are well suited to that location but user state would not be. A widget that is hidden may have it's renderer destroyed and if it is shown again the new renderer must be able to reflect the same widget state.

Using SimpleRenderer for custom widgets

Custom widgets built from a single `CanvasObject`, for example a container wrapping multiple builtin widgets, can be implemented using `SimpleRenderer`. Below example is a custom widget that can be used as item in a list view, showing a title on the left hand side that will be truncated when too long, and a comment on the right hand side. The constructor would be called from the `CreateItem` function of the list, and the title and comment changed in the `UpdateItem` function:

```
type MyListItemWidget struct {
        widget.BaseWidget
        Title   *widget.Label
        Comment *widget.Label
}

func NewMyListItemWidget(title, comment string) *MyListItemWidget {
        item := &MyListItemWidget{
                Title:   widget.NewLabel(title),
                Comment: widget.NewLabel(comment),
        }
        item.Title.Truncation = fyne.TextTruncateEllipsis
        item.ExtendBaseWidget(item)

        return item
}

func (item *MyListItemWidget) CreateRenderer() fyne.WidgetRenderer {
        c := container.NewBorder(nil, nil, nil, item.Comment, item.Title)
        return widget.NewSimpleRenderer(c)
}
```

Bundling resources

Go based applications are usually built as a single binary executable, and this is the same for Fyne applications. A single file makes it easier to distribute install our software. Unfortunately GUI applications typically require additional resources to render the user interface. To manage this challenge a Go application can bundle assets into the binary itself. The Fyne toolkit prefers the use of "fyne bundle" as it has various benefits that we will explore below.

The basic approach to bundle an asset is to execute the "fyne bundle" command. This tool has various parameters to customise the output, but in it's basic form the file to bundle will be converted into Go source code that can be built into your application.

```
$ ls
image.png       main.go
$ fyne bundle -o bundled.go image.png
$ ls
bundled.go      image.png       main.go
```

```
$
```

The contents of `bundled.go` will be a list of resource variables that we can then access in our code. For example the code above will result in a file that includes the following:

```
var resourceImagePng = &fyne.StaticResource{
        StaticName: "image.png",
        StaticContent: []byte{
...
        }}
```

As you see the default naming is “resource<Name>.<Ext>”. The name and package used in this file can be customised in command parameters. We can then use this name to, for example, load an image on our canvas:

```
img := canvas.NewImageFromResource(resourceImagePng)
```

A fyne resource is just a collection of bytes with a unique name, so this could be a font, a sound file or any other data you wish to load. Also you can bundle many resources into a single file using the `-append` parameter. If you will be bundling many files it is recommended to save the commands in a go:generate header in one of your go files (not bundled.go):

```
//go:generate fyne bundle -o bundled.go image1.png
//go:generate fyne bundle -o bundled.go -append image2.png
```

If you then change any assets or add new ones then you can update this header and run it with “go generate” to update your `bundled.go` file. You should then add `bundled.go` to version control so others can build your app without needing to run “fyne bundle”.

Creating a Custom Theme

Applications are able to load custom themes that can apply small changes to the standard theme or provide a completely unique look. A theme will need to implement the functions of `fyne.Theme` interface, which is defined as follows:

```
type Theme interface {
        Color(ThemeColorName, ThemeVariant) color.Color
        Font(TextStyle) Resource
        Icon(ThemeIconName) Resource
        Size(ThemeSizeName) float32
}
```

To apply our theme changes we will first define a new type that that implements this interface.

Define your theme

We start by defining a new type that will be our theme, a simple empty struct will do:

```
type myTheme struct {}
```

It is a good idea to assert that we implement an interface so that compile errors are closer to the defining type.

```
var _ fyne.Theme = (*myTheme)(nil)
```

At this point you could see compile errors because we still need to implement the methods, we start with color.

Customising colors

The `Color` function defined in the `Theme` interface asks us to define a named color and also provides a hint for the variant that the user desires (for example `theme.VariantLight` or `theme.VariantDark`). In our theme we will return a custom background color - using a different value for light and dark.

```
// The color package has to be imported from "image/color".

func (m myTheme) Color(name fyne.ThemeColorName, variant fyne.ThemeVariant) color.Color {
        if name == theme.ColorNameBackground {
                if variant == theme.VariantLight {
                        return color.White
                }
```

```
                return color.Black
        }

        return theme.DefaultTheme().Color(name, variant)
}
```

You will see the last line here references the `theme.DefaultTheme()` to look up standard values. This allows us to provide custom values and yet fall back to the standard theme when we don't want to provide our own values.

Of course colors are simpler than resources, we look at that for custom icons.

Overriding default icons

Icons (and Fonts) use `fyne.Resource` as values instead of simple types like `int` (for size) or `color.Color` for colors. We can build our own resource using `fyne.NewStaticResource`, or you could pass in a value that was created using resource embedding.

```
func (m myTheme) Icon(name fyne.ThemeIconName) fyne.Resource {
        if name == theme.IconNameHome {
                return fyne.NewStaticResource("myHome", homeBytes)
        }

        return theme.DefaultTheme().Icon(name)
}
```

As above we return the default theme icon if we don't want to provide a specific override.

Load the theme

Before we can load the theme you will need to implement the `Size` and `Font` methods as well. You can just use these empty implementations if you are happy to use the defaults.

```
func (m myTheme) Font(style fyne.TextStyle) fyne.Resource {
        return theme.DefaultTheme().Font(style)
}

func (m myTheme) Size(name fyne.ThemeSizeName) float32 {
        return theme.DefaultTheme().Size(name)
}
```

To set the theme for your app you will need to add the following line of code:

```
app.Settings().SetTheme(&myTheme{})
```

With these changes you can apply your own style, to make small tweaks or provide a completely custom looking application!

Extending Widgets

The standard Fyne widgets provide the minimum functionality and customisation to support most use-cases. It may be required at certain times to have more advanced functionality. Rather than have developers build their own widgets it is possible to extend the existing ones.

For example we will extend the icon widget to support being tapped. To do this we declare a new struct that embeds the `widget.Icon` type. We create a constructor function that calls the important `ExtendBaseWidget` function.

```
import (
        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/widget"
)

type tappableIcon struct {
        widget.Icon
}

func newTappableIcon(res fyne.Resource) *tappableIcon {
        icon := &tappableIcon{}
        icon.ExtendBaseWidget(icon)
        icon.SetResource(res)
```

```
        return icon
}
```

> **Note:** a widget constructor like `widget.NewIcon` may not be used for extension since it already calls `ExtendBaseWidget`.

We then add new functions to implement the `fyne.Tappable` interface, with those functions added the new `Tapped` function will be called every time the user taps our new icon. The interface required has just one function, `Tapped(*PointEvent)` and we implement it as follows:

```
import "log"

func (t *tappableIcon) Tapped(_ *fyne.PointEvent) {
        log.Println("I have been tapped")
}
```

We can test this new widget using a simple application as follows.

```
import (
    "fyne.io/fyne/v2/app"
    "fyne.io/fyne/v2/theme"
)

func main() {
        a := app.New()
        w := a.NewWindow("Tappable")
        w.SetContent(newTappableIcon(theme.FyneLogo()))
        w.ShowAndRun()
}
```

Numerical Entry

In the traditional sense, GUI programs have used callbacks to customize actions for widgets. Fyne does not expose inserting custom callbacks to capture events on widgets, but it does not need to. The Go language is plenty extensible to make this work.

Instead we can simply use Type Embedding and extend the widget to only make it possible to enter numerical values.

First create a new type struct, we will call it `numericalEntry`.

```
type numericalEntry struct {
    widget.Entry
}
```

As mentioned in Extending existing widgets, we follow good practice and create a constructor function that extends the `BaseWidget`.

```
func newNumericalEntry() *numericalEntry {
    entry := &numericalEntry{}
    entry.ExtendBaseWidget(entry)
    return entry
}
```

Now we need to make the entry accept only numbers. This can be done by overriding the `TypedRune(rune)` method that's part of the `fyne.Focusable` interface. This will allow us to intercept the standard handling of runes received from key presses and only pass through those that we want. Inside this method, we will use a conditional to check if the rune matches any of the numbers between zero and nine. If they do, we delegate it to the standard `TypedRune(rune)` method of the embedded entry. If they do not, we just ignore the inputs. This implementation will only allow integers to be entered, but can easily be extended to check for other keys in the future if necessary.

```
func (e *numericalEntry) TypedRune(r rune) {
        if r >= '0' && r <= '9' {
                e.Entry.TypedRune(r)
        }
}
```

If we want to update the implementation to allow for decimal numbers as well, we can simply add `.` and `,` to the list of allowed runes (some languages use commas over dots for decimal notations).

```
func (e *numericalEntry) TypedRune(r rune) {
        if (r >= '0' && r <= '9') || r == '.' || r == ',' {
                        e.Entry.TypedRune(r)
        }
}
```

With this, the entry now only allows the user to enter numerical values when keys are pressed. However, the paste shortcut will still allow text to be entered. To fix this, we can overwrite the `TypedShortcut(fyne.Shortcut)` method that is part of the `fyne.Shortcutable` interface. First we need to do a type assertion to check if the given shortcut is of the type `*fyne.ShortcutPaste`. If it is not, we can just delegate the shortcut back to the embedded entry. If it is, we check if the clipboard content is numerical, by using `strconv.ParseFloat()` (if you want to only allow integers, `strconv.Atoi()` will be just fine), and then delegating the shortcut back to the embedded entry if the clipboard content could be parsed without errors.

```
func (e *numericalEntry) TypedShortcut(shortcut fyne.Shortcut) {
        paste, ok := shortcut.(*fyne.ShortcutPaste)
        if !ok {
                e.Entry.TypedShortcut(shortcut)
                return
        }

        content := paste.Clipboard.Content()
        if _, err := strconv.ParseFloat(content, 64); err == nil {
                e.Entry.TypedShortcut(shortcut)
        }
}
```

As a bonus, we can also make sure that mobile operating systems open the numerical keyboard instead of the default keyboard. This can be done by by first importing the `fyne.io/fyne/v2/driver/mobile` package and overwriting the `Keyboard()` `mobile.KeyboardType` method that is part of the `mobile.Keyboardable` interface. Inside the function, we then simply return the `mobile.NumberKeyboard` type.

```
func (e *numericalEntry) Keyboard() mobile.KeyboardType {
        return mobile.NumberKeyboard
}
```

In the end, the resulting code could look something like this:

```
package main

import (
        "strconv"

        "fyne.io/fyne/v2"
        "fyne.io/fyne/v2/app"
        "fyne.io/fyne/v2/driver/mobile"
        "fyne.io/fyne/v2/widget"
)

type numericalEntry struct {
        widget.Entry
}

func newNumericalEntry() *numericalEntry {
        entry := &numericalEntry{}
        entry.ExtendBaseWidget(entry)
        return entry
}

func (e *numericalEntry) TypedRune(r rune) {
        if (r >= '0' && r <= '9') || r == '.' || r == ',' {
                e.Entry.TypedRune(r)
        }
}

func (e *numericalEntry) TypedShortcut(shortcut fyne.Shortcut) {
```

```
        paste, ok := shortcut.(*fyne.ShortcutPaste)
        if !ok {
                e.Entry.TypedShortcut(shortcut)
                return
        }

        content := paste.Clipboard.Content()
        if _, err := strconv.ParseFloat(content, 64); err == nil {
                e.Entry.TypedShortcut(shortcut)
        }
}

func (e *numericalEntry) Keyboard() mobile.KeyboardType {
        return mobile.NumberKeyboard
}

func main() {
        a := app.New()
        w := a.NewWindow("Numerical")

        entry := newNumericalEntry()

        w.SetContent(entry)
        w.ShowAndRun()
}
```

Geometry

Fyne apps are based on 1 canvas per window. Each canvas has a root CanvasObject which can be a single widget or a Container for many sub-objects whose size and position are controlled by a Layout.

Position

Each canvas has its origin at the top left (0, 0) every element of the UI may be scaled depending on the output device and so the API does not describe pixels or exact measurements. The position (10, 10) may be 10 pixels right and down from the origin on, for example, a 120DPI monitor but on a HiDPI (or “Retina”) display this will probably be closer to 20 pixels.

Every position referenced by a CanvasObject is relative to it’s parent. This is important for layout algorithms but also for developers in situations such as the `Tappable.Tapped(PointEvent)` handlers. Here the X/Y coordinates will be calculated from the top left of the button not the overall canvas. This is designed to allow code to be as self-contained as possible.

Pixel size

Like other vector based GUI libraries the Fyne coordinates need to be based on some baseline monitor resolution. All scaling is relative to this value. For fyne that resolution is 120DPI. This means that the sizes referred to in `fyne.Size` will be 1=1px when your monitor is 120DPI and scale values are all set to 1. For a HiDPI screen, as mentioned above, the actual DPI may be closer to 240 and on mobile devices it could even be 360 or higher. To manage handle this complexity the toolkit manages scaling internally so your apps will always look the right size. If a user sets the scale to be smaller then their apps will always have smaller than normal fonts, buttons etc, and when they specify larger then your app will scale up to suit.

When compared to Material Design we can see that their baseline DPI is 160, although the maths is similar the actual numbers will be different. This means that device-independent sizes in Fyne use a smaller number to represent the same physical size. For example an icon that is `18` tall in Fyne would be sized at `24` in a standard material design (for example Android) app. This does not matter when building your application, but may be important when working with designers or experts with Material Design.

One time that pixel sizes will matter is if you start loading bitmaps images. Normally these scale appropriately, but if you specify `FillMode=fyne.FillOriginal` then the actual image size will be different on different devices, due to the pixel density. Normally this feature would be used inside a `Scroll` container. Fyne also defines a `canvas.Raster` primitive which will draw pixels exactly at the pixel density of the output device. This enables your code to draw at the highest possible output resolution without knowing details of the device you are running on. If for some reason you need “pixel perfect” positioning you need to multiply `CanvasObject.Size()` by `Canvas.Scale()`.

Scaling

Fyne is built entirely using vector graphics, which means applications written with Fyne will scale to any size beautifully (not just whole number increments). This is a great benefit to the rising popularity of high density displays on mobile devices and high-end computers. The default scale value is calculated to match your operating system - on some systems this is user configuration and on others from your screen's pixel density (DPI - dots per inch). If a Fyne window is moved to another screen it will re-scale and adjust the window size accordingly! We call this "auto scaling", and it is designed to keep an app user interface the same size as you change monitor.

You can adjust the size of applications using the `fyne_settings` app or by setting a specific scale using the `FYNE_SCALE` environment variable. These values can make content larger or smaller than the system settings, using "1.5" will make things 50% larger or setting 0.8 will make it 20% smaller.

Widgets

Widgets in the Fyne toolkit are designed for a clean and pleasant user interaction, following a standard theme and supporting rapid app development, solid testing and easy maintenance. There are various design considerations that promote that ambition, we explore them in this page.

Behaviour API

One thing that you will notice about the standard widgets is that the API is all about behaviour and state - but very little that controls the actual look of an element. This is by design. It enables our code, and that of app developers, to focus on the behaviour of a widget so that it's rendering process can be left to other code. This makes it much easier to test, in fact full applications can be run through unit tests in memory without ever having to render the app.

You can extend an existing widget to add new behaviours without needing to worry about how it is rendered. It is also possible to write your own components, an application is not limited to using the provided widget set. When building your own widget you will notice that the rendering details are completely separate from the state - this is part of the design mentioned above. A `WidgetRenderer` (the code that renders a `Widget`) typically holds a reference to the widget that it will be rendering to access state or other information. When a widget state changes then `Refresh()` is called - the renderer will then be asked to refresh and it should update the display to reflect the new state. Custom widgets are recommended to use the current `Theme` but can choose to specify their own sizes, colours and icons where that seems desirable.

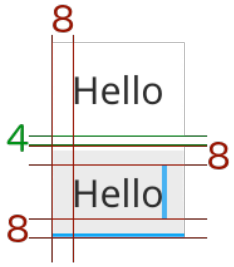
Content Padding

The standard widgets use the theme specified padding to make appropriate space around their graphical components. The `widget` package uses a standard height and baseline to ensure that provided layouts will align well by default. If you are building a custom widget it is recommended to follow these guidelines.

The value of `theme.Padding()` is used in layouts to space elements of a container, it creates a consistent space around the various parts of an application. Some widgets, however, have content that should be inset from the edges of the extents. Consider

`Entry`, It has a background and a border that go out to the edges, but its content should be inset. And so we have standardised the amount of spacing used to inset so that alignment matches.

The standard inset, or content padding, of a widget is defined as `theme.InnerPadding()`. The standard value of padding is `4` and the inner padding is `8`. You can see in `Label` and `Entry` how the (text) content is inset by this much so that their content will align horizontally and vertically when placed next to each other.



Organisation and Packages

The Fyne project is split into many packages, each providing different types of functionality. They are as follows:

`fyne.io/fyne/v2`
: This import provides the basic definitions common to all Fyne code including data types and interfaces.

`fyne.io/fyne/v2/app`
: The app package provides the APIs that start a new application. Normally you only require `app.New()` or `app.NewWithID()`.

`fyne.io/fyne/v2/canvas`
: The canvas package provides all of the drawing APIs within Fyne. The complete Fyne toolkit is made up of these primitive graphical types.

`fyne.io/fyne/v2/container`
: The container package provides containers that are used to lay out and organise applications.

`fyne.io/fyne/v2/data/binding`
: The binding package contains ways of binding data sources to widgets.

`fyne.io/fyne/v2/data/validation`
: The validation package provides tooling for validating data inside widgets.

`fyne.io/fyne/v2/dialog`
: The dialog package contains dialogs such as confirm, error and file save/open.

`fyne.io/fyne/v2/layout`
: The layout package provides various layout implementations for use with containers (discussed in a later tutorial).

`fyne.io/fyne/v2/storage`
: The storage package provides storage access and management functionality.

`fyne.io/fyne/v2/test`
: Applications can be tested more easily using the tools within the test package.

`fyne.io/fyne/v2/widget`
: Most graphical applications are created using a collection of widgets. All the widgets and interactive elements within Fyne are in this package.

Layout and Widget Size

Intro

Move and Resize

Q: How can I move my widget to a different position or resize it?

A: The position and size of elements in a Fyne app are controlled by the layout of the container that they are within. If the elements of your UI are too small consider using a different layout or container.

A new `Window` will expand whatever element is passed to `SetContent()` to fill its size. Each time you add a container to this it will divide up the available space according to the layout. Layouts like `HBox` and `VBox` will shrink content to its `MinSize()` in

one dimension or another to pack contents in. Layouts like `Max` or `Border` will expand content to fill the space. By writing a custom layout you could fully control the items within a container.

Q: Why is my image so small?

A: One of the difficulties in using a fully scalable user interface toolkit such as Fyne is that the coordinate system is device independent. This allows apps to draw at the right resolution or pixel density to get the best results based on the hardware connected. What this means for pixel based images is that their size could vary based on details not known at compile time.

Due to this complication an image loaded using `canvas.NewImageFromFile()` or similar calls will not have a size set, leading to it being very small or appearing to be hidden by default. When placed in an appropriate layout the image will stretch according to its `FillMode` property. If you desire the image to always be set to a certain size (or larger) you can call `Image.SetMinSize()` and specify a device independent size for the image.

Containers and Layout

Q: How can I manually control the position of elements

A: In some situations you may want to have complete control over the position and size of elements in a container. To do this you create a container without a layout. The `container.NewWithoutLayout()` function will create a container for manual positioning - you should pass to that constructor a list of the graphical elements that you want to manage in this container.

Once set up then you can use `Move()` and `Resize()` on each element to position it as desired. When doing this be careful to note that it will not adjust as the available space changes - and it does not have an explicit minimum size either. To add either of those features you should replace your manual positioning with a custom layout.

Theme and Customisation

In this page we answer some common questions about the design of Fyne themes and widgets.

Customisation

Q: How can I change the colour of text for a `Label` widget?

A: All of the standard widgets use the current `Theme` definition to set the colour, font and sizes. To make changes to your application consider using a custom theme.

If your application requires text that is a different colour you can use the `canvas.Text` type instead. That allows directly setting the colour and size of the text. Be careful when doing this because a user can choose between light or dark theme variations, so you should test with both.

Q: How can I remove the background colour from my `Entry` widget?

A: The input background is set by the theme `InputBackground` color. You can change that to `color.Transparent` to remove all input background boxes. It is not possible to edit the style of a single input element - the theme API is designed to give a customisable, but consistent, design.

Theme API

Q: How can I use my custom theme written before v2.0.0?

A: Over time you should consider updating to use the new theme API. However it is possible to use a simple adapter that was included to allow the usage of old themes during the transition time. You will find `theme.FromLegacy` function that can adapt an old theme instance to the new API.

```
myTheme := &myOldThemeType{}
updated := theme.FromLegacy(myTheme)
app.Settings().SetTheme(updated)
```

There are no performance penalties when using a theme in this mode, but in a future release this API will be removed.

Troubleshooting

Some things can not go as expected during setup or in compiling your first app. We try to address these problems here. Remember you can also check your configuration using the Fyne Setup tool.

Compiler Issues

Q: command not found: fyne

A: If you have installed the Fyne command using `go install fyne.io/tools/cmd/fyne@latest` but you see an error when trying to run it then the most likely issue is that your Go install has not set up your PATH environment variable correctly.

Go will install tools to the `bin` directory inside the user's `GOPATH` location (normally `~/go`) - you can check this by seeing if your `PATH` variable includes this location. If it seems to be missing then you should update your `PATH` environment variable to include `~/go/bin`, or if you have changed install location then you can use `$(go env GOPATH)/bin` instead.

Q: build constraints exclude all Go files in …

A: If you are cross compiling you may see an error about go files being excluded, followed by a build failure. When doing a standard Go cross-compile it will automatically turn off CGo. To fix this be sure to set `CGO_ENABLED=1` in your compile command.

Q: cc1.exe: sorry, unimplemented: 64-bit mode not compiled in

A: Windows compilation will sometimes complain that 64 bit mode is not available. This is normally because the wrong compiler is installed, or the configuration is incorrect. If you have followed our installation instructions for MSYS2 and MingW64 then make sure you are using the start menu launcher titled "MSYS2 MinGW 64-bit".

Distribution

Q: Apple macOS says my app is damaged when it is downloaded

A: When files are downloaded on a macOS computer they are marked with a "quarantine" flag so they are checked by the OS for problems. If your application is signed with a certificate purchased from Apple this is not a problem. However if you want to share your software without that cost this error may appear - and on M1/2 computers it is not possible to use the *System Settings* to allow the app to run.

The fix is to remove the quarantine flag, which you can do by opening the *Terminal* and executing the following command:

```
sudo xattr -r -d com.apple.quarantine MyApp.app
```

Upgrading to v2.6

The 2.6 release is fully backward compatible with 2.5.5 and earlier, however there are notable changes in how it works internally that may impact some apps.

Changes:

- All callbacks and widget functions are called on the same event thread
- It now matters whether your code runs in the background or not, see below
- Menu shortcuts will be called ahead of shortcuts on the current focused widget
- The fyne command-line tool has been moved to the fyne.io/tools project

Updating

Open your `go.mod` file and alter the the `require` line to use version `v2.6.0`, or you can execute the following command inside the directory:

```
go get fyne.io/fyne/v2@v2.6.0
```

The next time you build or run your app it will be using the 2.6 API, showing the new curved corners in input widgets and selection.

Fyne command

You should update the `fyne` tool for v2.6.0 to get some of the new features and bug fixes. This is now done by installing it from the tools repository. You can make the upgrade by using a `go install` command similar to above:

```
go install fyne.io/tools/cmd/fyne@latest
```

After that completes, check you have the new version installed by running `fyne version`.

Changes

Although this release is backwards compatible so your code will compile and run as expected, there are some changes which you may need to consider.

Thread handling

Fyne now uses a single thread / goroutine for handling all user events and data processing

This means that all callbacks will occur in serial, meaning there is a small possibility of user code having a deadlock that was not present before. If this occurs in your app double check that you are not holding internal locks (they should not be required for ensuring widget data safety any more).

Background access to Fyne APIs must be performed through fyne.Do[AndWait] functions

Because all Fyne callbacks now happen on a single thread (for speed and safety) apps using goroutines need to be updated. If your app launches a goroutine (using the `go` keyword) any Fyne APIs called from within the new context must be wrapped in a new accessor.

This new API has two varieties - `fyne.Do` and `fyne.DoAndWait`, for most situations the first will be best. The “AndWait” version should be used if your app must wait for the user interface to process the code passed in and update. If your change is a simple update and you can “fire and forget” simply call `fyne.Do`, the order of operations is guaranteed and changes will happen within the next graphical update.

Smaller changes

- Global shortcuts can now be registered by adding a menu item in the app’s MainMenu. Items in the menu will be actioned instead of sending the shortcut to a widget if there is a match. For a small number of apps this may cause a behaviour change - if so the menu item could check the current focused widget and forward the action.

Fyne API "fyne"

fyne

```
import "fyne.io/fyne/v2"
```

Package fyne describes the objects and components available to any Fyne app. These can all be created, manipulated and tested without rendering (for speed). Your main package should use the app package to create an application with a default driver that will render your UI.

A simple application may look like this:

```
    package main

    import "fyne.io/fyne/v2/app"
    import "fyne.io/fyne/v2/container"
    import "fyne.io/fyne/v2/widget"

    func main() {
        a := app.New()
        w := a.NewWindow("Hello")

        hello := widget.NewLabel("Hello Fyne!")
        w.SetContent(container.NewVBox(
                hello,
                widget.NewButton("Hi!", func() {
                        hello.SetText("Welcome :)")
                }),
        ))

        w.ShowAndRun()
    }
```

Usage

```
const AnimationRepeatForever = -1
```

AnimationRepeatForever is an AnimationCount value that indicates it should not stop looping.

Since: 2.0

```
const KeyModifierShortcutDefault = KeyModifierSuper
```

KeyModifierShortcutDefault is the default key modifier for shortcuts (Control or Command).

Since: 2.2

```
const KeyModifierShortcutDefault = KeyModifierControl
```

KeyModifierShortcutDefault is the default key modifier for shortcuts (Control or Command).

Since: 2.2

```
var (
        // AnimationEaseInOut is the default easing, it starts slowly, accelerates to the
middle and slows to the end.
        //
        // Since: 2.0
        AnimationEaseInOut = animationEaseInOut
        // AnimationEaseIn starts slowly and accelerates to the end.
        //
        // Since: 2.0
        AnimationEaseIn = animationEaseIn
        // AnimationEaseOut starts at speed and slows to the end.
        //
        // Since: 2.0
        AnimationEaseOut = animationEaseOut
        // AnimationLinear is a linear mapping for animations that progress uniformly
through their duration.
        //
        // Since: 2.0
        AnimationLinear = animationLinear
)
```

func Do

```
func Do(fn func())
```

Do is used to execute a specified function in the main Fyne runtime context without waiting. This is required when a background process wishes to adjust graphical elements of a running app. Developers should use this only from within goroutines they have created and when the result does not have to be waited for.

Since: 2.6

func DoAndWait

```
func DoAndWait(fn func())
```

DoAndWait is used to execute a specified function in the main Fyne runtime context. This is required when a background process wishes to adjust graphical elements of a running app. Developers should use this only from within goroutines they have created.

Since: 2.6

func IsHorizontal

```
func IsHorizontal(orient DeviceOrientation) bool
```

IsHorizontal is a helper utility that determines if a passed orientation is horizontal

func IsVertical

```
func IsVertical(orient DeviceOrientation) bool
```

IsVertical is a helper utility that determines if a passed orientation is vertical

func LogError

```
func LogError(reason string, err error)
```

LogError reports an error to the command line with the specified err cause, if not nil. The function also reports basic information about the code location.

func Max

```
func Max(x, y float32) float32
```

Max returns the larger of the passed values.

func Min

```
func Min(x, y float32) float32
```

Min returns the smaller of the passed values.

func SetCurrentApp

```
func SetCurrentApp(current App)
```

SetCurrentApp is an internal function to set the app instance currently running.

types

- Animation
- AnimationCurve
- App
- AppMetadata
- BuildType
- Canvas
- CanvasObject
- Clipboard
- CloudProvider
- CloudProviderPreferences
- CloudProviderStorage
- Container
- Delta
- Device
- DeviceOrientation
- Disableable
- DoubleTappable
- DragEvent
- Draggable
- Driver
- Focusable
- HardwareKey
- KeyEvent
- KeyModifier
- KeyName
- KeyboardShortcut
- Layout
- LegacyTheme
- Lifecycle
- ListableURI
- Locale
- MainMenu

- Menu
- MenuItem
- Notification
- OverlayStack
- PointEvent
- Position
- Preferences
- Resource
- ScrollDirection
- ScrollEvent
- Scrollable
- SecondaryTappable
- Settings
- Shortcut
- ShortcutCopy
- ShortcutCut
- ShortcutHandler
- ShortcutPaste
- ShortcutRedo
- ShortcutSelectAll
- ShortcutUndo
- Shortcutable
- Size
- StaticResource
- Storage
- StringValidator
- Tabbable
- Tappable
- TextAlign
- TextStyle
- TextTruncation
- TextWrap
- Theme
- ThemeColorName
- ThemeIconName
- ThemeSizeName
- ThemeVariant
- ThemedResource
- URI
- URIReadCloser
- URIWithIcon
- URIWriteCloser
- Validatable
- Vector2
- Widget
- WidgetRenderer
- Window

Fyne API "app"

app

```
import "fyne.io/fyne/v2/app"
```

Package app provides app implementations for working with Fyne graphical interfaces. The fastest way to get started is to call app.New() which will normally load a new desktop application. If the “ci” tag is passed to go (go run -tags ci myapp.go) it will run an in-memory application.

Usage

func New

```
func New() fyne.App
```

New returns a new application instance with the default driver and no unique ID (unless specified in FyneApp.toml)

func NewWithID

```
func NewWithID(id string) fyne.App
```

NewWithID returns a new app instance using the appropriate runtime driver. The ID string should be globally unique to this app.

func SetMetadata

```
func SetMetadata(m fyne.AppMetadata)
```

SetMetadata overrides the packaged application metadata. This data can be used in many places like notifications and about screens.

types

- SettingsSchema

Fyne API "canvas"

canvas

```
import "fyne.io/fyne/v2/canvas"
```

Package canvas contains all of the primitive CanvasObjects that make up a Fyne GUI.

The types implemented in this package are used as building blocks in order to build higher order functionality. These types are designed to be non-interactive, by design. If additional functionality is required, it’s usually a sign that this type should be used as part of a custom widget.

Usage

```
const (
        // DurationStandard is the time a standard interface animation will run.
        //
        // Since: 2.0
        DurationStandard = time.Millisecond * 300
        // DurationShort is the time a subtle or small transition should use.
        //
        // Since: 2.0
        DurationShort = time.Millisecond * 150
)
```

func NewColorRGBAAnimation

```
func NewColorRGBAAnimation(start, stop color.Color, d time.Duration, fn
func(color.Color)) *fyne.Animation
```

NewColorRGBAAnimation sets up a new animation that will transition from the start to stop Color over the specified Duration. The colour transition will move linearly through the RGB colour space. The content of fn should apply the color values to an object and refresh it. You should call Start() on the returned animation to start it.

Since: `2.0`

func NewPositionAnimation

```
func NewPositionAnimation(start, stop fyne.Position, d time.Duration, fn
func(fyne.Position)) *fyne.Animation
```

NewPositionAnimation sets up a new animation that will transition from the start to stop Position over the specified Duration. The content of fn should apply the position value to an object for the change to be visible. You should call Start() on the returned animation to start it.

Since: `2.0`

func NewSizeAnimation

```
func NewSizeAnimation(start, stop fyne.Size, d time.Duration, fn func(fyne.Size))
*fyne.Animation
```

NewSizeAnimation sets up a new animation that will transition from the start to stop Size over the specified Duration. The content of fn should apply the size value to an object for the change to be visible. You should call Start() on the returned animation to start it.

Since: `2.0`

func RecolorSVG

```
func RecolorSVG(svgContent []byte, color color.Color) ([]byte, error)
```

RecolorSVG takes a []byte containing SVG content, and returns new SVG content, re-colorized to be monochrome with the given color. The content can be assigned to a new fyne.StaticResource with an appropriate name to be used in a widget.Button, canvas.Image, etc.

If an error occurs, the returned content will be the original un-modified content, and a non-nil error is returned.

Since: `2.6`

func Refresh

```
func Refresh(obj fyne.CanvasObject)
```

Refresh instructs the containing canvas to refresh the specified obj.

types

- Circle
- Image
- ImageFill
- ImageScale
- Line
- LinearGradient
- RadialGradient
- Raster
- Rectangle
- Text

Fyne API "container"

container

```
import "fyne.io/fyne/v2/container"
```

Package container provides containers that are used to lay out and organise applications.

Usage

```
const (
        // ScrollBoth supports horizontal and vertical scrolling.
        ScrollBoth ScrollDirection = fyne.ScrollBoth
        // ScrollHorizontalOnly specifies the scrolling should only happen left to right.
        ScrollHorizontalOnly = fyne.ScrollHorizontalOnly
        // ScrollVerticalOnly specifies the scrolling should only happen top to bottom.
        ScrollVerticalOnly = fyne.ScrollVerticalOnly
        // ScrollNone turns off scrolling for this container.
        //
        // Since: 2.1
        ScrollNone = fyne.ScrollNone
)
```

Constants for valid values of ScrollDirection.

func New

```
func New(layout fyne.Layout, objects ...fyne.CanvasObject) *fyne.Container
```

New returns a new Container instance holding the specified CanvasObjects which will be laid out according to the specified Layout.

Since: `2.0`

func NewAdaptiveGrid

```
func NewAdaptiveGrid(rowcols int, objects ...fyne.CanvasObject) *fyne.Container
```

NewAdaptiveGrid creates a new container with the specified objects and using the grid layout. When in a horizontal arrangement the rowcols parameter will specify the column count, when in vertical it will specify the rows. On mobile this will dynamically refresh when device is rotated.

Since: `1.4`

func NewBorder

```
func NewBorder(top, bottom, left, right fyne.CanvasObject, objects ...fyne.CanvasObject)
*fyne.Container
```

NewBorder creates a new container with the specified objects and using the border layout. The top, bottom, left and right parameters specify the items that should be placed around edges. Nil can be used to an edge if it should not be filled. Passed objects not assigned to any edge (parameters 5 onwards) will be used to fill the space remaining in the middle. Parameters 6 onwards will be stacked over the middle content in the specified order as a Stack container.

Since: `1.4`

func NewCenter

```
func NewCenter(objects ...fyne.CanvasObject) *fyne.Container
```

NewCenter creates a new container with the specified objects centered in the available space.

Since: `1.4`

func NewGridWithColumns

```
func NewGridWithColumns(cols int, objects ...fyne.CanvasObject) *fyne.Container
```

NewGridWithColumns creates a new container with the specified objects and using the grid layout with a specified number of columns. The number of rows will depend on how many children are in the container.

Since: `1.4`

func NewGridWithRows

```
func NewGridWithRows(rows int, objects ...fyne.CanvasObject) *fyne.Container
```

NewGridWithRows creates a new container with the specified objects and using the grid layout with a specified number of rows. The number of columns will depend on how many children are in the container.

Since: 1.4

func NewGridWrap

```
func NewGridWrap(size fyne.Size, objects ...fyne.CanvasObject) *fyne.Container
```

NewGridWrap creates a new container with the specified objects and using the gridwrap layout. Every element will be resized to the size parameter and the content will arrange along a row and flow to a new row if the elements don't fit.

Since: 1.4

func NewHBox

```
func NewHBox(objects ...fyne.CanvasObject) *fyne.Container
```

NewHBox creates a new container with the specified objects and using the HBox layout. The objects will be placed in the container from left to right and always displayed at their horizontal MinSize. Use a different layout if the objects are intended to be larger then their horizontal MinSize.

Since: 1.4

func NewMax

```
func NewMax(objects ...fyne.CanvasObject) *fyne.Container
```

NewMax creates a new container with the specified objects filling the available space.

Since: 1.4
Deprecated: Use container.NewStack() instead.

func NewPadded

```
func NewPadded(objects ...fyne.CanvasObject) *fyne.Container
```

NewPadded creates a new container with the specified objects inset by standard padding size.

Since: 1.4

func NewStack

```
func NewStack(objects ...fyne.CanvasObject) *fyne.Container
```

NewStack returns a new container that stacks objects on top of each other. Objects at the end of the container will be stacked on top of objects before. Having only a single object has no impact as CanvasObjects will fill the available space even without a Stack.

Since: 2.4

func NewVBox

```
func NewVBox(objects ...fyne.CanvasObject) *fyne.Container
```

NewVBox creates a new container with the specified objects and using the VBox layout. The objects will be stacked in the container from top to bottom and always displayed at their vertical MinSize. Use a different layout if the objects are intended to be larger then their vertical MinSize.

Since: 1.4

func NewWithoutLayout

```
func NewWithoutLayout(objects ...fyne.CanvasObject) *fyne.Container
```

NewWithoutLayout returns a new Container instance holding the specified CanvasObjects that are manually arranged.

Since: 2.0

types

- AppTabs
- DocTabs
- InnerWindow
- MultipleWindows
- Scroll
- ScrollDirection
- Split
- TabItem
- TabLocation
- ThemeOverride

Fyne API "binding"

binding

```
import "fyne.io/fyne/v2/data/binding"
```

Package binding provides support for binding data to widgets. All APIs in the binding package are safe to invoke directly from any goroutine.

Usage

```
const DataTreeRootID = ""
```

DataTreeRootID const is the value used as ID for the root of any tree binding.

types

- Bool
- BoolList
- BoolTree
- Bytes
- BytesList
- BytesTree
- DataItem
- DataList
- DataListener
- DataMap
- DataTree
- ExternalBool
- ExternalBoolList
- ExternalBoolTree
- ExternalBytes
- ExternalBytesList
- ExternalBytesTree
- ExternalFloat
- ExternalFloatList
- ExternalFloatTree
- ExternalInt
- ExternalIntList
- ExternalIntTree
- ExternalItem
- ExternalRune
- ExternalRuneList

- ExternalRuneTree
- ExternalString
- ExternalStringList
- ExternalStringTree
- ExternalURI
- ExternalURIList
- ExternalURITree
- ExternalUntyped
- ExternalUntypedList
- ExternalUntypedMap
- ExternalUntypedTree
- Float
- FloatList
- FloatTree
- Int
- IntList
- IntTree
- Item
- Rune
- RuneList
- RuneTree
- String
- StringList
- StringTree
- Struct
- URI
- URIList
- URITree
- Untyped
- UntypedList
- UntypedMap
- UntypedTree

Fyne API "validation"

validation

```
import "fyne.io/fyne/v2/data/validation"
```

Package validation provides validation for data inside widgets.

Usage

func NewAllStrings

```
func NewAllStrings(validators ...fyne.StringValidator) fyne.StringValidator
```

NewAllStrings creates a validator that requires all of the passed string validators to pass. In short, it combines multiple string validators into one.

Since: `2.2`

func NewRegexp

```
func NewRegexp(regexpstr, reason string) fyne.StringValidator
```

NewRegexp creates a new validator that uses regular expression parsing. The validator will return nil if valid, otherwise returns an error with a reason text.

Since: 1.4

func NewTime

```
func NewTime(format string) fyne.StringValidator
```

NewTime creates a new validator that verifies times using time.Parse. The validator will return nil if valid, otherwise returns an error with a reason text. The reference time for the format: Mon Jan 2 15:04:05 -0700 MST 2006. See time.Parse() for more information about the reference time: https://pkg.go.dev/time#Parse

Since: 2.1

types

Fyne API "dialog"

dialog

```
import "fyne.io/fyne/v2/dialog"
```

Package dialog defines standard dialog windows for application GUIs.

Usage

func ShowColorPicker

```
func ShowColorPicker(title, message string, callback func(c color.Color), parent
fyne.Window)
```

ShowColorPicker creates and shows a color dialog. The callback is triggered when the user selects a color.

Since: 1.4

func ShowConfirm

```
func ShowConfirm(title, message string, callback func(bool), parent fyne.Window)
```

ShowConfirm shows a dialog over the specified window for a user confirmation. The title is used for the dialog window and message is the content. The callback is executed when the user decides.

func ShowCustom

```
func ShowCustom(title, dismiss string, content fyne.CanvasObject, parent fyne.Window)
```

ShowCustom shows a dialog over the specified application using custom content. The button will have the dismiss text set. The MinSize() of the CanvasObject passed will be used to set the size of the window.

func ShowCustomConfirm

```
func ShowCustomConfirm(title, confirm, dismiss string, content fyne.CanvasObject,
        callback func(bool), parent fyne.Window)
```

ShowCustomConfirm shows a dialog over the specified application using custom content. The cancel button will have the dismiss text set and the “OK” will use the confirm text. The response callback is called on user action. The MinSize() of the CanvasObject passed will be used to set the size of the window.

func ShowCustomWithoutButtons

```
func ShowCustomWithoutButtons(title string, content fyne.CanvasObject, parent
fyne.Window)
```

ShowCustomWithoutButtons shows a dialog, without buttons, over the specified application using custom content. The MinSize() of the CanvasObject passed will be used to set the size of the window.

Since: 2.4

func ShowEntryDialog

```
func ShowEntryDialog(title, message string, onConfirm func(string), parent fyne.Window)
```

ShowEntryDialog creates a new entry dialog and shows it immediately.

Deprecated: Use dialog.ShowForm() with a widget.Entry inside instead.

func ShowError

```
func ShowError(err error, parent fyne.Window)
```

ShowError shows a dialog over the specified window for an application error. The message is extracted from the provided error (should not be nil).

func ShowFileOpen

```
func ShowFileOpen(callback func(reader fyne.URIReadCloser, err error), parent
fyne.Window)
```

ShowFileOpen creates and shows a file dialog allowing the user to choose a file to open.

The callback function will run when the dialog closes and provide a reader for the chosen file. The reader will be nil when the user cancels or when nothing is selected. When the reader isn't nil it must be closed by the callback.

The dialog will appear over the window specified.

func ShowFileSave

```
func ShowFileSave(callback func(writer fyne.URIWriteCloser, err error), parent
fyne.Window)
```

ShowFileSave creates and shows a file dialog allowing the user to choose a file to save to (new or overwrite). If the user chooses an existing file they will be asked if they are sure.

The callback function will run when the dialog closes and provide a writer for the chosen file. The writer will be nil when the user cancels or when nothing is selected. When the writer isn't nil it must be closed by the callback.

The dialog will appear over the window specified.

func ShowFolderOpen

```
func ShowFolderOpen(callback func(fyne.ListableURI, error), parent fyne.Window)
```

ShowFolderOpen creates and shows a file dialog allowing the user to choose a folder to open. The callback function will run when the dialog closes. The URI will be nil when the user cancels or when nothing is selected.

The dialog will appear over the window specified.

Since: 1.4

func ShowForm

```
func ShowForm(title, confirm, dismiss string, content []*widget.FormItem, callback
func(bool), parent fyne.Window)
```

ShowForm shows a dialog over the specified application using the provided FormItems. The cancel button will have the dismiss text set and the confirm button will use the confirm text. The response callback is called on user action after validation passes. If any Validatable widget reports that validation has failed, then the confirm button will be disabled. The initial state of the confirm button will reflect the initial validation state of the items added to the form dialog. The MinSize() of the CanvasObject passed will be used to set the size of the window.

Since: 2.0

func ShowInformation

```
func ShowInformation(title, message string, parent fyne.Window)
```

ShowInformation shows a dialog over the specified window for user information. The title is used for the dialog window and message is the content.

types

- ColorPickerDialog
- ConfirmDialog
- CustomDialog
- Dialog
- EntryDialog
- FileDialog
- FormDialog
- ProgressDialog
- ProgressInfiniteDialog
- ViewLayout

Fyne API "driver"

driver

```
import "fyne.io/fyne/v2/driver"
```

Usage

func RunNative

```
func RunNative(fn func(any) error) error
```

RunNative provides a way to execute code within the platform-specific runtime context for various runtimes. On Android this provides the JVM pointers required to execute various NDK calls or use JNI APIs.

Since: `2.3`

types

- AndroidContext
- AndroidWindowContext
- MacWindowContext
- NativeWindow
- UnknownContext
- WaylandWindowContext
- WindowsWindowContext
- X11WindowContext

Fyne API "desktop"

desktop

```
import "fyne.io/fyne/v2/driver/desktop"
```

Package desktop provides desktop specific driver functionality.

Usage

```
const (
        // KeyNone represents no key
        KeyNone fyne.KeyName = ""
        // KeyShiftLeft represents the left shift key
        KeyShiftLeft fyne.KeyName = "LeftShift"
        // KeyShiftRight represents the right shift key
        KeyShiftRight fyne.KeyName = "RightShift"
        // KeyControlLeft represents the left control key
        KeyControlLeft fyne.KeyName = "LeftControl"
        // KeyControlRight represents the right control key
        KeyControlRight fyne.KeyName = "RightControl"
        // KeyAltLeft represents the left alt key
        KeyAltLeft fyne.KeyName = "LeftAlt"
        // KeyAltRight represents the right alt key
        KeyAltRight fyne.KeyName = "RightAlt"
        // KeySuperLeft represents the left "Windows" key (or "Command" key on macOS)
        KeySuperLeft fyne.KeyName = "LeftSuper"
        // KeySuperRight represents the right "Windows" key (or "Command" key on macOS)
        KeySuperRight fyne.KeyName = "RightSuper"
        // KeyMenu represents the left or right menu / application key
        KeyMenu fyne.KeyName = "Menu"
        // KeyPrintScreen represents the key used to cause a screen capture
        KeyPrintScreen fyne.KeyName = "PrintScreen"

        // KeyCapsLock represents the caps lock key, tapping once is the down event then
again is the up
        KeyCapsLock fyne.KeyName = "CapsLock"
)

const (
        // ShiftModifier represents a shift key being held
        //
        // Deprecated: Use fyne.KeyModifierShift instead.
        ShiftModifier = fyne.KeyModifierShift
        // ControlModifier represents the ctrl key being held
        //
        // Deprecated: Use fyne.KeyModifierControl instead.
        ControlModifier = fyne.KeyModifierControl
        // AltModifier represents either alt keys being held
        //
        // Deprecated: Use fyne.KeyModifierAlt instead.
        AltModifier = fyne.KeyModifierAlt
        // SuperModifier represents either super keys being held
        //
        // Deprecated: Use fyne.KeyModifierSuper instead.
        SuperModifier = fyne.KeyModifierSuper
)
```

types

- App
- Canvas
- Cursor
- Cursorable
- CustomShortcut
- Driver
- Hoverable
- Keyable
- Modifier
- MouseButton
- MouseEvent
- Mouseable
- StandardCursor

Fyne API "mobile"

mobile

```
import "fyne.io/fyne/v2/driver/mobile"
```

Package mobile provides mobile specific driver functionality.

Package mobile provides desktop specific mobile functionality.

Usage

```
const (
        // KeyBack represents the back button which may be hardware or software
        KeyBack fyne.KeyName = "Back"
)
```

types

- Device
- Driver
- KeyboardType
- Keyboardable
- TouchEvent
- Touchable

Fyne API "software"

software

```
import "fyne.io/fyne/v2/driver/software"
```

Usage

func NewCanvas

```
func NewCanvas() test.WindowlessCanvas
```

NewCanvas creates a new canvas in memory that can render without hardware support.

func NewTransparentCanvas

```
func NewTransparentCanvas() test.WindowlessCanvas
```

NewTransparentCanvas creates a new canvas in memory that can render without hardware support without a background color.

Since: `2.2`

func Render

```
func Render(obj fyne.CanvasObject, t fyne.Theme) image.Image
```

Render takes a canvas object and renders it to a regular Go image using the provided Theme. The returned image will be set to the object's minimum size. Use the theme.LightTheme() or theme.DarkTheme() to access the builtin themes.

func RenderCanvas

```
func RenderCanvas(c fyne.Canvas, t fyne.Theme) image.Image
```

RenderCanvas takes a canvas and renders it to a regular Go image using the provided Theme. This is the same as setting the application theme and then calling Canvas.Capture().

types

Fyne API "lang"

lang

```
import "fyne.io/fyne/v2/lang"
```

Package lang introduces a translation and localisation API for Fyne applications

Since 2.5

Usage

```
var (
        // L is a shortcut to localize a string, similar to the gettext "_" function.
        // More info available on the `Localize` function.
        L = Localize

        // N is a shortcut to localize a string with plural forms, similar to the
ngettext function.
        // More info available on the `LocalizePlural` function.
        N = LocalizePlural

        // X is a shortcut to get the localization of a string with specified key,
similar to pgettext.
        // More info available on the `LocalizeKey` function.
        X = LocalizeKey

        // XN is a shortcut to get the localization plural form of a string with
specified key, similar to npgettext.
        // More info available on the `LocalizePluralKey` function.
        XN = LocalizePluralKey
)
```

func AddTranslations

```
func AddTranslations(r fyne.Resource) error
```

AddTranslations allows an app to load a bundle of translations. The language that this relates to will be inferred from the resource name, for example “fr.json”. The data should be in json format.

func AddTranslationsFS

```
func AddTranslationsFS(fs embed.FS, dir string) (retErr error)
```

AddTranslationsFS supports adding all translations in one calling using an `embed.FS` setup. The `dir` parameter specifies the name or path of the directory containing translation files inside this embedded filesystem. Each file should be a json file with the name following pattern [prefix.]lang.json.

func AddTranslationsForLocale

```
func AddTranslationsForLocale(data []byte, l fyne.Locale) error
```

AddTranslationsForLocale allows an app to load a bundle of translations for a specified locale. The data should be in json format.

func Localize

```
func Localize(in string, data ...any) string
```

Localize asks the translation engine to translate a string, this behaves like the gettext "_" function. The string can be templated and the template data can be passed as a struct with exported fields, or as a map of string keys to any suitable value.

func LocalizeKey

```
func LocalizeKey(key, fallback string, data ...any) string
```

LocalizeKey asks the translation engine for the translation with specific ID. If it cannot be found then the fallback will be used. The string can be templated and the template data can be passed as a struct with exported fields, or as a map of string keys to any suitable value.

func LocalizePlural

```
func LocalizePlural(in string, count int, data ...any) string
```

LocalizePlural asks the translation engine to translate a string from one of a number of plural forms. This behaves like the ngettext function, with the `count` parameter determining the plurality looked up. The string can be templated and the template data can be passed as a struct with exported fields, or as a map of string keys to any suitable value.

func LocalizePluralKey

```
func LocalizePluralKey(key, fallback string, count int, data ...any) string
```

LocalizePluralKey asks the translation engine for the translation with specific ID in plural form. This behaves like the npgettext function, with the `count` parameter determining the plurality looked up. If it cannot be found then the fallback will be used. The string can be templated and the template data can be passed as a struct with exported fields, or as a map of string keys to any suitable value.

func SystemLocale

```
func SystemLocale() fyne.Locale
```

SystemLocale returns the primary locale on the current system. This may refer to a language that Fyne does not have translations for.

types

Fyne API "layout"

layout

```
import "fyne.io/fyne/v2/layout"
```

Package layout defines the various layouts available to Fyne apps.

Usage

func NewAdaptiveGridLayout

```
func NewAdaptiveGridLayout(rowcols int) fyne.Layout
```

NewAdaptiveGridLayout returns a new grid layout which uses columns when horizontal but rows when vertical.

func NewBorderLayout

```
func NewBorderLayout(top, bottom, left, right fyne.CanvasObject) fyne.Layout
```

NewBorderLayout creates a new BorderLayout instance with top, bottom, left and right objects set. All other items in the container will fill the remaining space in the middle. Multiple extra items will be stacked in the specified order as a Stack container.

func NewCenterLayout

```
func NewCenterLayout() fyne.Layout
```

NewCenterLayout creates a new CenterLayout instance

func NewCustomPaddedHBoxLayout

```
func NewCustomPaddedHBoxLayout(padding float32) fyne.Layout
```

NewCustomPaddedHBoxLayout returns a layout similar to HBoxLayout that uses a custom amount of padding in between objects instead of the theme.Padding value.

Since: 2.5

func NewCustomPaddedLayout

```
func NewCustomPaddedLayout(padTop, padBottom, padLeft, padRight float32) fyne.Layout
```

NewCustomPaddedLayout creates a new CustomPaddedLayout instance with the specified paddings.

Since: 2.5

func NewCustomPaddedVBoxLayout

```
func NewCustomPaddedVBoxLayout(padding float32) fyne.Layout
```

NewCustomPaddedVBoxLayout returns a layout similar to VBoxLayout that uses a custom amount of padding in between objects instead of the theme.Padding value.

Since: 2.5

func NewFormLayout

```
func NewFormLayout() fyne.Layout
```

NewFormLayout returns a new FormLayout instance

func NewGridLayout

```
func NewGridLayout(cols int) fyne.Layout
```

NewGridLayout returns a grid layout arranged in a specified number of columns. The number of rows will depend on how many children are in the container that uses this layout.

func NewGridLayoutWithColumns

```
func NewGridLayoutWithColumns(cols int) fyne.Layout
```

NewGridLayoutWithColumns returns a new grid layout that specifies a column count and wrap to new rows when needed.

func NewGridLayoutWithRows

```
func NewGridLayoutWithRows(rows int) fyne.Layout
```

NewGridLayoutWithRows returns a new grid layout that specifies a row count that creates new rows as required.

func NewGridWrapLayout

```
func NewGridWrapLayout(size fyne.Size) fyne.Layout
```

NewGridWrapLayout returns a new GridWrapLayout instance

func NewHBoxLayout

```
func NewHBoxLayout() fyne.Layout
```

NewHBoxLayout returns a horizontal box layout for stacking a number of child canvas objects or widgets left to right. The objects are always displayed at their horizontal MinSize. Use a different layout if the objects are intended to be larger than their horizontal MinSize.

func NewMaxLayout

```
func NewMaxLayout() fyne.Layout
```

NewMaxLayout creates a new MaxLayout instance

Deprecated: Use layout.NewStackLayout() instead.

func NewPaddedLayout

```
func NewPaddedLayout() fyne.Layout
```

NewPaddedLayout creates a new PaddedLayout instance

Since: 1.4

func NewSpacer

```
func NewSpacer() fyne.CanvasObject
```

NewSpacer returns a spacer object which can fill vertical and horizontal space. This is primarily used with a box layout.

func NewStackLayout

```
func NewStackLayout() fyne.Layout
```

NewStackLayout returns a new StackLayout instance. Objects are stacked on top of each other with later objects on top of those before. Having only a single object has no impact as CanvasObjects will fill the available space even without a Stack.

Since: 2.4

func NewVBoxLayout

```
func NewVBoxLayout() fyne.Layout
```

NewVBoxLayout returns a vertical box layout for stacking a number of child canvas objects or widgets top to bottom. The objects are always displayed at their vertical MinSize. Use a different layout if the objects are intended to be larger than their vertical MinSize.

types

- CustomPaddedLayout
- Spacer
- SpacerObject

Fyne API "storage"

storage

```
import "fyne.io/fyne/v2/storage"
```

Package storage provides storage access and management functionality.

Usage

```
var (
        // ErrAlreadyExists may be thrown by docs. E.g., save a document twice.
        //
        // Since: 2.3
        ErrAlreadyExists = errors.New("document already exists")

        // ErrNotExists may be thrown by docs. E.g., save an unknown document.
        //
        // Since: 2.3
```

```
        ErrNotExists = errors.New("document does not exist")
)

var URIRootError = repository.ErrURIRoot
```

URIRootError is a wrapper for repository.URIRootError

Deprecated - use repository.ErrURIRoot instead

func Appender

```
func Appender(u fyne.URI) (fyne.URIWriteCloser, error)
```

Appender returns URIWriteCloser set up to write to the resource that the URI references without truncating it first

Writing to a non-extant resource should create that resource if possible (and if not possible, this should be reflected in the return of CanWrite()). Writing to an extant resource should NOT overwrite it in-place.

This method can fail in several ways:

```
    - Different permissions or credentials are required to write to the
      referenced resource.

    - This URI scheme could represent some resources that can be
      written, but this particular URI references a resources that is
      not something that can be written.

    - Attempting to set up the writer depended on a lower level
      operation such as a network or filesystem access that has failed
      in some way.

    - If the scheme of the given URI does not have a registered
      AppendableRepository instance, then this method will fail with a
      repository.ErrOperationNotSupported.
```

Appender is backed by the repository system - this function calls into a scheme-specific implementation from a registered repository.

Since: `2.6`

func CanList

```
func CanList(u fyne.URI) (bool, error)
```

CanList will determine if the URI is listable or not.

This method may fail in several ways:

```
    - Different permissions or credentials are required to check if the
      URI supports listing.

    - This URI scheme could represent some resources that can be listed,
      but this specific URI is not one of them (e.g. a file on a
      filesystem, as opposed to a directory).

    - Checking for listability depended on a lower level operation
      such as network or filesystem access that has failed in some way.

    - If the scheme of the given URI does not have a registered
      ListableRepository instance, then this method will fail with a
      repository.ErrOperationNotSupported.
```

CanList is backed by the repository system - this function calls into a scheme-specific implementation from a registered repository.

Since: `2.0`

func CanRead

```
func CanRead(u fyne.URI) (bool, error)
```

CanRead determines if a given URI could be written to using the Reader() method. It is preferred to check if a URI is readable using this method before calling Reader(), because the underlying operations required to attempt to read and then report an error may be slower than the operations needed to test if a URI is readable. Keep in mind however that even if CanRead returns true, you must still do appropriate error handling for Reader(), as the underlying filesystem may have changed since you called CanRead.

The non-existence of a resource should not be treated as an error. In other words, a Repository implementation which for some URI u returns false, nil for Exists(u), CanRead(u) should also return false, nil.

CanRead is backed by the repository system - this function calls into a scheme-specific implementation from a registered repository.

Since: `2.0`

func CanWrite

```
func CanWrite(u fyne.URI) (bool, error)
```

CanWrite is backed by the repository system - this function calls into a scheme-specific implementation from a registered repository.

Since: `2.0`

func Child

```
func Child(u fyne.URI, component string) (fyne.URI, error)
```

Child returns a URI referencing a resource nested hierarchically below the given URI, identified by a string. For example, the child with the string component ‘quux’ of ‘file://foo/bar’ is ‘file://foo/bar/quux’.

This can fail in several ways:

```
- If the URI refers to a resource which does not exist in a hierarchical
  context (e.g. the URI references something which does not have a
  semantically meaningful "child"), the Child() implementation may return an
  error.

- If generating a reference to a child of the referenced resource requires
  interfacing with some external system, failures may propagate through the
  Child() implementation. It is expected that this case would occur very
  rarely if ever.

- If the scheme of the given URI does not have a registered
  HierarchicalRepository instance, then this method will fail with a
  repository.ErrOperationNotSupported.
```

NOTE: since v2.0.0, Child() is backed by the repository system - this function is a helper which calls into an appropriate repository instance for the scheme of the URI it is given.

Since: `1.4`

func Copy

```
func Copy(source fyne.URI, destination fyne.URI) error
```

Copy given two URIs, ‘src’, and ‘dest’ both of the same scheme, will copy one to the other. If the source and destination are of different schemes, then the Copy implementation for the storage repository registered to the scheme of the source will be used. Implementations are recommended to use repository.GenericCopy() as a fail-over in the case that they do not understand how to operate on the scheme of the destination URI. However, the behavior of calling Copy() on URIs of non-matching schemes is ultimately defined by the storage repository registered to the scheme of the source URI.

This method may fail in several ways:

```
- Different permissions or credentials are required to perform the
  copy operation.

- This URI scheme could represent some resources that can be copied,
  but either the source, destination, or both are not resources
  that support copying.

- Performing the copy operation depended on a lower level operation
  such as network or filesystem access that has failed in some way.
```

```
- If the scheme of the given URI does not have a registered
  CopyableRepository instance, then this method will fail with a
  repository.ErrOperationNotSupported.
```

Copy is backed by the repository system - this function calls into a scheme-specific implementation from a registered repository.

Since: 2.0

func CreateListable

```
func CreateListable(u fyne.URI) error
```

CreateListable creates a new listable resource referenced by the given URI. CreateListable will error if the URI already references an extant resource. This method is used for storage repositories where listable resources are of a different underlying type than other resources - for example, in a typical filesystem (‘file://’), CreateListable() corresponds to directory creation, and Writer() implies file creation for non-extant operands.

For storage repositories where listable and non-listable resources are the of the same underlying type, CreateListable should be equivalent to calling Writer(), writing zero bytes, and then closing the `URIWriteCloser - in filesystem terms, the same as calling ‘touch;’.

Storage repositories which support listing, but not creation of listable objects may return repository.ErrOperationNotSupported.

CreateListable should generally fail if the parent of its operand does not exist, however this can vary by the implementation details of the specific storage repository. In filesystem terms, this function is “mkdir” not “mkdir -p”.

This method may fail in several ways:

```
- Different permissions or credentials are required to create the requested
  resource.

- Creating the resource depended on a lower level operation such as network
  or filesystem access that has failed in some way.

- If the scheme of the given URI does not have a registered
  ListableRepository instance, then this method will fail with a
  repository.ErrOperationNotSupported.
```

CreateListable is backed by the repository system - this function either calls into a scheme-specific implementation from a registered repository, or fails with a URIOperationNotSupported error.

Since: 2.0

func Delete

```
func Delete(u fyne.URI) error
```

Delete destroys, deletes, or otherwise removes the resource referenced by the URI.

This can fail in several ways:

```
- If removing the resource requires interfacing with some external system,
  failures may propagate through Destroy(). For example, deleting a file may
  fail with a permissions error.

- If the referenced resource does not exist, attempting to destroy it should
  throw an error.

- If the scheme of the given URI does not have a registered
  WritableRepository instance, then this method will fail with a
  repository.ErrOperationNotSupported.
```

Delete is backed by the repository system - this function calls into a scheme-specific implementation from a registered repository.

Since: 2.0

func EqualURI

```
func EqualURI(t1, t2 fyne.URI) bool
```

EqualURI returns true if the two URIs are equal.

Since: 2.6

func Exists

```
func Exists(u fyne.URI) (bool, error)
```

Exists determines if the resource referenced by the URI exists.

This can fail in several ways:

- If checking the existence of a resource requires interfacing with some external system, then failures may propagate through Exists(). For example, checking the existence of a resource requires reading a directory may result in a permissions error.

It is understood that a non-nil error value signals that the existence or non-existence of the resource cannot be determined and is undefined.

NOTE: since v2.0.0, Exists is backed by the repository system - this function calls into a scheme-specific implementation from a registered repository.

Exists may call into either a generic implementation, or into a scheme-specific implementation depending on which storage repositories have been registered.

Since: 1.4

func List

```
func List(u fyne.URI) ([]fyne.URI, error)
```

List returns a list of URIs that reference resources which are nested below the resource referenced by the argument. For example, listing a directory on a filesystem should return a list of files and directories it contains.

This method may fail in several ways:

- Different permissions or credentials are required to obtain a listing for the given URI.

- This URI scheme could represent some resources that can be listed, but this specific URI is not one of them (e.g. a file on a filesystem, as opposed to a directory). This can be tested in advance using the Listable() function.

- Obtaining the listing depended on a lower level operation such as network or filesystem access that has failed in some way.

- If the scheme of the given URI does not have a registered ListableRepository instance, then this method will fail with a repository.ErrOperationNotSupported.

List is backed by the repository system - this function either calls into a scheme-specific implementation from a registered repository, or fails with a URIOperationNotSupported error.

Since: 2.0

func ListerForURI

```
func ListerForURI(uri fyne.URI) (fyne.ListableURI, error)
```

ListerForURI will attempt to use the application’s driver to convert a standard URI into a listable URI.

Since: 1.4

func LoadResourceFromURI

```
func LoadResourceFromURI(u fyne.URI) (fyne.Resource, error)
```

LoadResourceFromURI creates a new StaticResource in memory using the contents of the specified URI. The URI will be opened using the current driver, so valid schemas will vary from platform to platform. The file:// schema will always work.

func Move

```
func Move(source fyne.URI, destination fyne.URI) error
```

Move returns a method that given two URIs, ‘src’ and ‘dest’ both of the same scheme this will move src to dest. This means the resource referenced by src will be copied into the resource referenced by dest, and the resource referenced by src will no longer exist after the operation is complete.

If the source and destination are of different schemes, then the Move implementation for the storage repository registered to the scheme of the source will be used. Implementations are recommended to use repository.GenericMove() as a fail-over in the case that they do not understand how to operate on the scheme of the destination URI. However, the behavior of calling Move() on URIs of non-matching schemes is ultimately defined by the storage repository registered to the scheme of the source URI.

This method may fail in several ways:

- Different permissions or credentials are required to perform the rename operation.
- This URI scheme could represent some resources that can be renamed, but either the source, destination, or both are not resources that support renaming.
- Performing the rename operation depended on a lower level operation such as network or filesystem access that has failed in some way.
- If the scheme of the given URI does not have a registered MovableRepository instance, then this method will fail with a repository.ErrOperationNotSupported.

Move is backed by the repository system - this function calls into a scheme-specific implementation from a registered repository.

Since: 2.0

func NewFileURI

```
func NewFileURI(path string) fyne.URI
```

NewFileURI creates a new URI from the given file path. Relative paths will be converted to absolute using filepath.Abs if required.

func NewURI

```
func NewURI(s string) fyne.URI
```

NewURI creates a new URI from the given string representation. This could be a URI from an external source or one saved from URI.String()

Deprecated: use ParseURI instead

func OpenFileFromURI

```
func OpenFileFromURI(uri fyne.URI) (fyne.URIReadCloser, error)
```

OpenFileFromURI loads a file read stream from a resource identifier. This is mostly provided so that file references can be saved using their URI and loaded again later.

Deprecated: this has been replaced by storage.Reader(URI)

func Parent

```
func Parent(u fyne.URI) (fyne.URI, error)
```

Parent returns a URI referencing the parent resource of the resource referenced by the URI. For example, the Parent() of ‘file://foo/bar.baz’ is ‘file://foo’. The URI which is returned will be listable.

NOTE: it is not a given that Parent() return a parent URI with the same Scheme(), though this will normally be the case.

This can fail in several ways:

```
- If the URI refers to a filesystem root, then the Parent() implementation
  must return (nil, URIRootError).

- If the URI refers to a resource which does not exist in a hierarchical
  context (e.g. the URI references something which does not have a
  semantically meaningful "parent"), the Parent() implementation may return
  an error.

- If determining the parent of the referenced resource requires
  interfacing with some external system, failures may propagate
  through the Parent() implementation. For example if determining
  the parent of a file:// URI requires reading information from
  the filesystem, it could fail with a permission error.

- If the scheme of the given URI does not have a registered
  HierarchicalRepository instance, then this method will fail with a
  repository.ErrOperationNotSupported.
```

NOTE: since v2.0.0, Parent() is backed by the repository system - this function is a helper which calls into an appropriate repository instance for the scheme of the URI it is given.

Since: `1.4`

func ParseURI

```
func ParseURI(s string) (fyne.URI, error)
```

ParseURI creates a new URI instance by parsing a URI string.

Parse URI will parse up to the first ‘:’ present in the URI string to extract the scheme, and then delegate further parsing to the registered repository for the given scheme. If no repository is registered for that scheme, the URI is parsed on a best-effort basis using net/url.

As a special exception, URIs beginning with ‘file:’ are always parsed using NewFileURI(), which will correctly handle back-slashes appearing in the URI path component on Windows.

Since: `2.0`

func Reader

```
func Reader(u fyne.URI) (fyne.URIReadCloser, error)
```

Reader returns URIReadCloser set up to read from the resource that the URI references.

This method can fail in several ways:

```
- Different permissions or credentials are required to read the
  referenced resource.

- This URI scheme could represent some resources that can be read,
  but this particular URI references a resources that is not
  something that can be read.

- Attempting to set up the reader depended on a lower level
  operation such as a network or filesystem access that has failed
  in some way.
```

Reader is backed by the repository system - this function calls into a scheme-specific implementation from a registered repository.

Since: `2.0`

func SaveFileToURI

```
func SaveFileToURI(uri fyne.URI) (fyne.URIWriteCloser, error)
```

SaveFileToURI loads a file write stream to a resource identifier. This is mostly provided so that file references can be saved using their URI and written to again later.

Deprecated: this has been replaced by storage.Writer(URI)

func Writer

```
func Writer(u fyne.URI) (fyne.URIWriteCloser, error)
```

Writer returns URIWriteCloser set up to write to the resource that the URI references.

Writing to a non-extant resource should create that resource if possible (and if not possible, this should be reflected in the return of CanWrite()). Writing to an extant resource should overwrite it in-place. At present, this API does not provide a mechanism for appending to an already-extant resource, except for reading it in and writing all the data back out.

This method can fail in several ways:

```
    - Different permissions or credentials are required to write to the
      referenced resource.

    - This URI scheme could represent some resources that can be
      written, but this particular URI references a resources that is
      not something that can be written.

    - Attempting to set up the writer depended on a lower level
      operation such as a network or filesystem access that has failed
      in some way.

    - If the scheme of the given URI does not have a registered
      WritableRepository instance, then this method will fail with a
      repository.ErrOperationNotSupported.
```

Writer is backed by the repository system - this function calls into a scheme-specific implementation from a registered repository.

Since: `2.0`

types

- ExtensionFileFilter
- FileFilter
- MimeTypeFileFilter

Fyne API "repository"

repository

```
import "fyne.io/fyne/v2/storage/repository"
```

Package repository provides primitives for working with storage repositories.

Usage

```
var (
        // ErrOperationNotSupported may be thrown by certain functions in the storage
        // or repository packages which operate on URIs if an operation is attempted
        // that is not supported for the scheme relevant to the URI, normally because
        // the underlying repository has either not implemented the relevant function,
        // or has explicitly returned this error.
        //
        // Since: 2.0
        ErrOperationNotSupported = errors.New("operation not supported for this URI")
```

```
        // ErrURIRoot should be thrown by fyne.URI implementations when the caller
        // attempts to take the parent of the root. This way, downstream code that
        // wants to programmatically walk up a URIs parent's will know when to stop
        // iterating.
        //
        // Since: 2.0
        ErrURIRoot = errors.New("cannot take the parent of the root element in a URI")
)
```

func EqualURI

```
func EqualURI(t1, t2 fyne.URI) bool
```

EqualURI returns true if the two URIs are equal.

Since: 2.6

func GenericChild

```
func GenericChild(u fyne.URI, component string) (fyne.URI, error)
```

GenericChild can be used as a common-case implementation of HierarchicalRepository.Child(). It will create a child URI by separating the URI into its component parts as described in IETF RFC 3986, then appending “/” + component to the path, then concatenating the result and parsing it as a new URI.

NOTE: this function should not be called except by an implementation of the Repository interface - using this for unknown URIs may break.

Since: 2.0

func GenericCopy

```
func GenericCopy(source fyne.URI, destination fyne.URI) error
```

GenericCopy can be used a common-case implementation of CopyableRepository.Copy(). It will perform the copy by obtaining a reader for the source URI, a writer for the destination URI, then writing the contents of the source to the destination.

For obvious reasons, the destination URI must have a registered WritableRepository.

NOTE: this function should not be called except by an implementation of the Repository interface - using this for unknown URIs may break.

Since: 2.0

func GenericMove

```
func GenericMove(source fyne.URI, destination fyne.URI) error
```

GenericMove can be used a common-case implementation of MovableRepository.Move(). It will perform the move by obtaining a reader for the source URI, a writer for the destination URI, then writing the contents of the source to the destination. Following this, the source will be deleted using WritableRepository.Delete.

For obvious reasons, the source and destination URIs must both be writable.

NOTE: this function should not be called except by an implementation of the Repository interface - using this for unknown URIs may break.

Since: 2.0

func GenericParent

```
func GenericParent(u fyne.URI) (fyne.URI, error)
```

GenericParent can be used as a common-case implementation of HierarchicalRepository.Parent(). It will create a parent URI based on IETF RFC3986.

In short, the URI is separated into its component parts, the path component is split along instances of ‘/’, and the trailing element is removed. The result is concatenated and parsed as a new URI.

If the URI path is empty or ‘/’, then a nil URI is returned, along with ErrURIRoot.

NOTE: this function should not be called except by an implementation of the Repository interface - using this for unknown URIs may break.

Since: `2.0`

func NewFileURI

```
func NewFileURI(path string) fyne.URI
```

NewFileURI implements the back-end logic to storage.NewFileURI, which you should use instead. This is only here because other functions in repository need to call it, and it prevents a circular import.

Since: `2.0`

func ParseURI

```
func ParseURI(s string) (fyne.URI, error)
```

ParseURI implements the back-end logic for storage.ParseURI, which you should use instead. This is only here because other functions in repository need to call it, and it prevents a circular import.

Since: `2.0`

func Register

```
func Register(scheme string, repository Repository)
```

Register registers a storage repository so that operations on URIs of the registered scheme will use methods implemented by the relevant repository implementation.

Since: `2.0`

types

- AppendableRepository
- CopyableRepository
- CustomURIRepository
- HierarchicalRepository
- ListableRepository
- MovableRepository
- Repository
- WritableRepository

Fyne API "test"

test

```
import "fyne.io/fyne/v2/test"
```

Package test provides utility drivers for running UI tests without rendering to a screen.

Usage

func ApplyTheme

```
func ApplyTheme(t *testing.T, theme fyne.Theme)
```

ApplyTheme sets the given theme and waits for it to be applied to the current app.

func AssertAllColorNamesDefined

```
func AssertAllColorNamesDefined(t *testing.T, th fyne.Theme, themeName string)
```

AssertAllColorNamesDefined asserts that all known color names are defined for the given theme.

func AssertCanvasTappableAt

```
func AssertCanvasTappableAt(t *testing.T, c fyne.Canvas, pos fyne.Position) bool
```

AssertCanvasTappableAt asserts that the canvas is tappable at the given position.

func AssertImageMatches

```
func AssertImageMatches(t *testing.T, masterFilename string, img image.Image,
msgAndArgs ...any) bool
```

AssertImageMatches asserts that the given image is the same as the one stored in the master file. The master filename is relative to the `testdata` directory which is relative to the test. The test `t` fails if the given image is not equal to the loaded master image. In this case the given image is written into a file in `testdata/failed/<masterFilename>` (relative to the test). This path is also reported, thus the file can be used as new master.

func AssertNotificationSent

```
func AssertNotificationSent(t *testing.T, n *fyne.Notification, f func())
```

AssertNotificationSent allows an app developer to assert that a notification was sent. After the content of f has executed this utility will check that the specified notification was sent.

func AssertObjectRendersToImage

```
func AssertObjectRendersToImage(t *testing.T, masterFilename string, o fyne.CanvasObject,
msgAndArgs ...any) bool
```

AssertObjectRendersToImage asserts that the given `CanvasObject` renders the same image as the one stored in the master file. The theme used is the standard test theme which may look different to how it shows on your device. The master filename is relative to the `testdata` directory which is relative to the test. The test `t` fails if the given image is not equal to the loaded master image. In this case the given image is written into a file in `testdata/failed/<masterFilename>` (relative to the test). This path is also reported, thus the file can be used as new master.

Since 2.3

func AssertObjectRendersToMarkup

```
func AssertObjectRendersToMarkup(t *testing.T, masterFilename string, o
fyne.CanvasObject, msgAndArgs ...any) bool
```

AssertObjectRendersToMarkup asserts that the given `CanvasObject` renders the same markup as the one stored in the master file. The master filename is relative to the `testdata` directory which is relative to the test. The test `t` fails if the rendered markup is not equal to the loaded master markup. In this case the rendered markup is written into a file in `testdata/failed/<masterFilename>` (relative to the test). This path is also reported, thus the file can be used as new master.

Be aware, that the indentation has to use tab characters (‘\t’) instead of spaces. Every element starts on a new line indented one more than its parent. Closing elements stand on their own line, too, using the same indentation as the opening element. The only exception to this are text elements which do not contain line breaks unless the text includes them.

Since 2.3

func AssertRendersToImage

```
func AssertRendersToImage(t *testing.T, masterFilename string, c fyne.Canvas,
msgAndArgs ...any) bool
```

AssertRendersToImage asserts that the given canvas renders the same image as the one stored in the master file. The master filename is relative to the `testdata` directory which is relative to the test. The test `t` fails if the given image is not equal to the loaded master image. In this case the given image is written into a file in `testdata/failed/<masterFilename>` (relative to the test). This path is also reported, thus the file can be used as new master.

Since 2.3

func AssertRendersToMarkup

```
func AssertRendersToMarkup(t *testing.T, masterFilename string, c fyne.Canvas, msgAndArgs ...any) bool
```

AssertRendersToMarkup asserts that the given canvas renders the same markup as the one stored in the master file. The master filename is relative to the `testdata` directory which is relative to the test. The test `t` fails if the rendered markup is not equal to the loaded master markup. In this case the rendered markup is written into a file in `testdata/failed/<masterFilename>` (relative to the test). This path is also reported, thus the file can be used as new master.

Be aware, that the indentation has to use tab characters (‘\t’) instead of spaces. Every element starts on a new line indented one more than its parent. Closing elements stand on their own line, too, using the same indentation as the opening element. The only exception to this are text elements which do not contain line breaks unless the text includes them.

Since: `2.0`

func Canvas

```
func Canvas() fyne.Canvas
```

Canvas returns a reusable in-memory canvas used for testing

func DoubleTap

```
func DoubleTap(obj fyne.DoubleTappable)
```

DoubleTap simulates a double left mouse click on the specified object.

func Drag

```
func Drag(c fyne.Canvas, pos fyne.Position, deltaX, deltaY float32)
```

Drag drags at an absolute position on the canvas. deltaX/Y is the dragging distance: <0 for dragging up/left, >0 for dragging down/right.

func FocusNext

```
func FocusNext(c fyne.Canvas)
```

FocusNext focuses the next focusable on the canvas.

func FocusPrevious

```
func FocusPrevious(c fyne.Canvas)
```

FocusPrevious focuses the previous focusable on the canvas.

func KnownThemeVariants

```
func KnownThemeVariants() map[string]fyne.ThemeVariant
```

KnownThemeVariants returns the known theme variants mapped by a descriptive key.

func LaidOutObjects

```
func LaidOutObjects(o fyne.CanvasObject) (objects []fyne.CanvasObject)
```

LaidOutObjects returns all fyne.CanvasObject starting at the given fyne.CanvasObject which is laid out previously.

func MoveMouse

```
func MoveMouse(c fyne.Canvas, pos fyne.Position)
```

MoveMouse simulates a mouse movement to the given position.

func NewApp

```
func NewApp() fyne.App
```

NewApp returns a new dummy app used for testing. It loads a test driver which creates a virtual window in memory for testing.

func NewClipboard

```
func NewClipboard() fyne.Clipboard
```

NewClipboard returns a single use in-memory clipboard used for testing

func NewDriver

```
func NewDriver() fyne.Driver
```

NewDriver sets up and registers a new dummy driver for test purpose

func NewDriverWithPainter

```
func NewDriverWithPainter(painter SoftwarePainter) fyne.Driver
```

NewDriverWithPainter creates a new dummy driver that will pass the given painter to all canvases created

func NewTempApp

```
func NewTempApp(t testing.TB) fyne.App
```

NewTempApp returns a new dummy app and tears it down at the end of the test.

Since: `2.5`

func NewTempWindow

```
func NewTempWindow(t testing.TB, content fyne.CanvasObject) fyne.Window
```

NewTempWindow creates and registers a new window for test purposes. This window will get removed automatically once the running test ends.

Since: `2.5`

func NewTheme

```
func NewTheme() fyne.Theme
```

NewTheme returns a new test theme using quiet ugly colors.

func NewWindow

```
func NewWindow(content fyne.CanvasObject) fyne.Window
```

NewWindow creates and registers a new window for test purposes

func RenderObjectToMarkup

```
func RenderObjectToMarkup(o fyne.CanvasObject) string
```

RenderObjectToMarkup renders the given [fyne.io/fyne/v2.CanvasObject] to a markup string.

Since: `2.6`

func RenderToMarkup

```
func RenderToMarkup(c fyne.Canvas) string
```

RenderToMarkup renders the given [fyne.io/fyne/v2.Canvas] to a markup string.

Since: 2.6

func Scroll

```
func Scroll(c fyne.Canvas, pos fyne.Position, deltaX, deltaY float32)
```

Scroll scrolls at an absolute position on the canvas. deltaX/Y is the scrolling distance: <0 for scrolling up/left, >0 for scrolling down/right.

func Tap

```
func Tap(obj fyne.Tappable)
```

Tap simulates a left mouse click on the specified object.

func TapAt

```
func TapAt(obj fyne.Tappable, pos fyne.Position)
```

TapAt simulates a left mouse click on the passed object at a specified place within it.

func TapCanvas

```
func TapCanvas(c fyne.Canvas, pos fyne.Position)
```

TapCanvas taps at an absolute position on the canvas.

func TapSecondary

```
func TapSecondary(obj fyne.SecondaryTappable)
```

TapSecondary simulates a right mouse click on the specified object.

func TapSecondaryAt

```
func TapSecondaryAt(obj fyne.SecondaryTappable, pos fyne.Position)
```

TapSecondaryAt simulates a right mouse click on the passed object at a specified place within it.

func TempWidgetRenderer

```
func TempWidgetRenderer(t *testing.T, wid fyne.Widget) fyne.WidgetRenderer
```

TempWidgetRenderer allows test scripts to gain access to the current renderer for a widget. This can be used for verifying correctness of rendered components for a widget in unit tests. The widget renderer is automatically destroyed when the test ends.

Since: 2.5

func Theme

```
func Theme() fyne.Theme
```

Theme returns a test theme useful for image based tests.

func Type

```
func Type(obj fyne.Focusable, chars string)
```

Type performs a series of key events to simulate typing of a value into the specified object. The focusable object will be focused before typing begins. The chars parameter will be input one rune at a time to the focused object.

func TypeOnCanvas

```
func TypeOnCanvas(c fyne.Canvas, chars string)
```

TypeOnCanvas is like the Type function but it passes the key events to the canvas object rather than a focusable widget.

func WidgetRenderer

```
func WidgetRenderer(wid fyne.Widget) fyne.WidgetRenderer
```

WidgetRenderer allows test scripts to gain access to the current renderer for a widget. This can be used for verifying correctness of rendered components for a widget in unit tests.

func WithTestTheme

```
func WithTestTheme(t *testing.T, f func())
```

WithTestTheme runs a function with the testTheme temporarily set.

types

- SoftwarePainter
- WindowlessCanvas

Fyne API "theme"

theme

```
import "fyne.io/fyne/v2/theme"
```

Package theme defines how a Fyne app should look when rendered.

Usage

```
const (
        // ColorRed is the red primary color name.
        //
        // Since: 1.4
        ColorRed = internaltheme.ColorRed
        // ColorOrange is the orange primary color name.
        //
        // Since: 1.4
        ColorOrange = internaltheme.ColorOrange
        // ColorYellow is the yellow primary color name.
        //
        // Since: 1.4
        ColorYellow = internaltheme.ColorYellow
        // ColorGreen is the green primary color name.
        //
        // Since: 1.4
        ColorGreen = internaltheme.ColorGreen
        // ColorBlue is the blue primary color name.
        //
        // Since: 1.4
        ColorBlue = internaltheme.ColorBlue
        // ColorPurple is the purple primary color name.
        //
        // Since: 1.4
        ColorPurple = internaltheme.ColorPurple
        // ColorBrown is the brown primary color name.
        //
        // Since: 1.4
        ColorBrown = internaltheme.ColorBrown
        // ColorGray is the gray primary color name.
        //
```

```
	// Since: 1.4
	ColorGray = internaltheme.ColorGray

	// ColorNameBackground is the name of theme lookup for background color.
	//
	// Since: 2.0
	ColorNameBackground fyne.ThemeColorName = "background"

	// ColorNameButton is the name of theme lookup for button color.
	//
	// Since: 2.0
	ColorNameButton fyne.ThemeColorName = "button"

	// ColorNameDisabledButton is the name of theme lookup for disabled button color.
	//
	// Since: 2.0
	ColorNameDisabledButton fyne.ThemeColorName = "disabledButton"

	// ColorNameDisabled is the name of theme lookup for disabled foreground color.
	//
	// Since: 2.0
	ColorNameDisabled fyne.ThemeColorName = "disabled"

	// ColorNameError is the name of theme lookup for error color.
	//
	// Since: 2.0
	ColorNameError fyne.ThemeColorName = "error"

	// ColorNameFocus is the name of theme lookup for focus color.
	//
	// Since: 2.0
	ColorNameFocus fyne.ThemeColorName = "focus"

	// ColorNameForeground is the name of theme lookup for foreground color.
	//
	// Since: 2.0
	ColorNameForeground fyne.ThemeColorName = "foreground"

	// ColorNameForegroundOnError is the name of theme lookup for a contrast color to
the error color.
	//
	// Since: 2.5
	ColorNameForegroundOnError fyne.ThemeColorName = "foregroundOnError"

	// ColorNameForegroundOnPrimary is the name of theme lookup for a contrast color
to the primary color.
	//
	// Since: 2.5
	ColorNameForegroundOnPrimary fyne.ThemeColorName = "foregroundOnPrimary"

	// ColorNameForegroundOnSuccess is the name of theme lookup for a contrast color
to the success color.
	//
	// Since: 2.5
	ColorNameForegroundOnSuccess fyne.ThemeColorName = "foregroundOnSuccess"

	// ColorNameForegroundOnWarning is the name of theme lookup for a contrast color
to the warning color.
	//
	// Since: 2.5
	ColorNameForegroundOnWarning fyne.ThemeColorName = "foregroundOnWarning"

	// ColorNameHeaderBackground is the name of theme lookup for background color of
a collection header.
	//
	// Since: 2.4
	ColorNameHeaderBackground fyne.ThemeColorName = "headerBackground"

	// ColorNameHover is the name of theme lookup for hover color.
	//
	// Since: 2.0
```

```
        ColorNameHover fyne.ThemeColorName = "hover"

        // ColorNameHyperlink is the name of theme lookup for hyperlink color.
        //
        // Since: 2.4
        ColorNameHyperlink fyne.ThemeColorName = "hyperlink"

        // ColorNameInputBackground is the name of theme lookup for background color of
an input field.
        //
        // Since: 2.0
        ColorNameInputBackground fyne.ThemeColorName = "inputBackground"

        // ColorNameInputBorder is the name of theme lookup for border color of an input
field.
        //
        // Since: 2.3
        ColorNameInputBorder fyne.ThemeColorName = "inputBorder"

        // ColorNameMenuBackground is the name of theme lookup for background color of
menus.
        //
        // Since: 2.3
        ColorNameMenuBackground fyne.ThemeColorName = "menuBackground"

        // ColorNameOverlayBackground is the name of theme lookup for background color of
overlays like dialogs.
        //
        // Since: 2.3
        ColorNameOverlayBackground fyne.ThemeColorName = "overlayBackground"

        // ColorNamePlaceHolder is the name of theme lookup for placeholder text color.
        //
        // Since: 2.0
        ColorNamePlaceHolder fyne.ThemeColorName = "placeholder"

        // ColorNamePressed is the name of theme lookup for the tap overlay color.
        //
        // Since: 2.0
        ColorNamePressed fyne.ThemeColorName = "pressed"

        // ColorNamePrimary is the name of theme lookup for primary color.
        //
        // Since: 2.0
        ColorNamePrimary fyne.ThemeColorName = "primary"

        // ColorNameScrollBar is the name of theme lookup for scrollbar color.
        //
        // Since: 2.0
        ColorNameScrollBar fyne.ThemeColorName = "scrollBar"

        // ColorNameScrollBarBackground is the name of theme lookup for scrollbar
background color.
        //
        // Since: 2.6
        ColorNameScrollBarBackground fyne.ThemeColorName = "scrollBarBackground"

        // ColorNameSelection is the name of theme lookup for selection color.
        //
        // Since: 2.1
        ColorNameSelection fyne.ThemeColorName = "selection"

        // ColorNameSeparator is the name of theme lookup for separator bars.
        //
        // Since: 2.3
        ColorNameSeparator fyne.ThemeColorName = "separator"

        // ColorNameShadow is the name of theme lookup for shadow color.
        //
        // Since: 2.0
        ColorNameShadow fyne.ThemeColorName = "shadow"
```

```
        // ColorNameSuccess is the name of theme lookup for success color.
        //
        // Since: 2.3
        ColorNameSuccess fyne.ThemeColorName = "success"

        // ColorNameWarning is the name of theme lookup for warning color.
        //
        // Since: 2.3
        ColorNameWarning fyne.ThemeColorName = "warning"
)
```

Keep in mind to add new constants to the tests at test/theme.go.

```
const (
        // IconNameCancel is the name of theme lookup for cancel icon.
        //
        // Since: 2.0
        IconNameCancel fyne.ThemeIconName = "cancel"

        // IconNameConfirm is the name of theme lookup for confirm icon.
        //
        // Since: 2.0
        IconNameConfirm fyne.ThemeIconName = "confirm"

        // IconNameDelete is the name of theme lookup for delete icon.
        //
        // Since: 2.0
        IconNameDelete fyne.ThemeIconName = "delete"

        // IconNameSearch is the name of theme lookup for search icon.
        //
        // Since: 2.0
        IconNameSearch fyne.ThemeIconName = "search"

        // IconNameSearchReplace is the name of theme lookup for search and replace icon.
        //
        // Since: 2.0
        IconNameSearchReplace fyne.ThemeIconName = "searchReplace"

        // IconNameMenu is the name of theme lookup for menu icon.
        //
        // Since: 2.0
        IconNameMenu fyne.ThemeIconName = "menu"

        // IconNameMenuExpand is the name of theme lookup for menu expansion icon.
        //
        // Since: 2.0
        IconNameMenuExpand fyne.ThemeIconName = "menuExpand"

        // IconNameCheckButton is the name of theme lookup for unchecked check button
icon.
        //
        // Since: 2.0
        IconNameCheckButton fyne.ThemeIconName = "unchecked"

        // IconNameCheckButtonChecked is the name of theme lookup for checked check
button icon.
        //
        // Since: 2.0
        IconNameCheckButtonChecked fyne.ThemeIconName = "checked"

        // IconNameCheckButtonFill is the name of theme lookup for filled check button
icon.
        //
        // Since: 2.5
        IconNameCheckButtonFill fyne.ThemeIconName = "iconNameCheckButtonFill"

        // IconNameCheckButtonPartial is the name of theme lookup for "partially" checked
check button icon.
        //
```

```
        // Since: 2.6
        IconNameCheckButtonPartial fyne.ThemeIconName = "partial"

        // IconNameRadioButton is the name of theme lookup for radio button unchecked
icon.
        //
        // Since: 2.0
        IconNameRadioButton fyne.ThemeIconName = "radioButton"

        // IconNameRadioButtonChecked is the name of theme lookup for radio button
checked icon.
        //
        // Since: 2.0
        IconNameRadioButtonChecked fyne.ThemeIconName = "radioButtonChecked"

        // IconNameRadioButtonFill is the name of theme lookup for filled radio button
icon.
        //
        // Since: 2.5
        IconNameRadioButtonFill fyne.ThemeIconName = "iconNameRadioButtonFill"

        // IconNameColorAchromatic is the name of theme lookup for greyscale color icon.
        //
        // Since: 2.0
        IconNameColorAchromatic fyne.ThemeIconName = "colorAchromatic"

        // IconNameColorChromatic is the name of theme lookup for full color icon.
        //
        // Since: 2.0
        IconNameColorChromatic fyne.ThemeIconName = "colorChromatic"

        // IconNameColorPalette is the name of theme lookup for color palette icon.
        //
        // Since: 2.0
        IconNameColorPalette fyne.ThemeIconName = "colorPalette"

        // IconNameContentAdd is the name of theme lookup for content add icon.
        //
        // Since: 2.0
        IconNameContentAdd fyne.ThemeIconName = "contentAdd"

        // IconNameContentRemove is the name of theme lookup for content remove icon.
        //
        // Since: 2.0
        IconNameContentRemove fyne.ThemeIconName = "contentRemove"

        // IconNameContentCut is the name of theme lookup for content cut icon.
        //
        // Since: 2.0
        IconNameContentCut fyne.ThemeIconName = "contentCut"

        // IconNameContentCopy is the name of theme lookup for content copy icon.
        //
        // Since: 2.0
        IconNameContentCopy fyne.ThemeIconName = "contentCopy"

        // IconNameContentPaste is the name of theme lookup for content paste icon.
        //
        // Since: 2.0
        IconNameContentPaste fyne.ThemeIconName = "contentPaste"

        // IconNameContentClear is the name of theme lookup for content clear icon.
        //
        // Since: 2.0
        IconNameContentClear fyne.ThemeIconName = "contentClear"

        // IconNameContentRedo is the name of theme lookup for content redo icon.
        //
        // Since: 2.0
        IconNameContentRedo fyne.ThemeIconName = "contentRedo"
```

```
	// IconNameContentUndo is the name of theme lookup for content undo icon.
	//
	// Since: 2.0
	IconNameContentUndo fyne.ThemeIconName = "contentUndo"

	// IconNameInfo is the name of theme lookup for info icon.
	//
	// Since: 2.0
	IconNameInfo fyne.ThemeIconName = "info"

	// IconNameQuestion is the name of theme lookup for question icon.
	//
	// Since: 2.0
	IconNameQuestion fyne.ThemeIconName = "question"

	// IconNameWarning is the name of theme lookup for warning icon.
	//
	// Since: 2.0
	IconNameWarning fyne.ThemeIconName = "warning"

	// IconNameError is the name of theme lookup for error icon.
	//
	// Since: 2.0
	IconNameError fyne.ThemeIconName = "error"

	// IconNameBrokenImage is the name of the theme lookup for broken-image icon.
	//
	// Since: 2.4
	IconNameBrokenImage fyne.ThemeIconName = "broken-image"

	// IconNameDocument is the name of theme lookup for document icon.
	//
	// Since: 2.0
	IconNameDocument fyne.ThemeIconName = "document"

	// IconNameDocumentCreate is the name of theme lookup for document create icon.
	//
	// Since: 2.0
	IconNameDocumentCreate fyne.ThemeIconName = "documentCreate"

	// IconNameDocumentPrint is the name of theme lookup for document print icon.
	//
	// Since: 2.0
	IconNameDocumentPrint fyne.ThemeIconName = "documentPrint"

	// IconNameDocumentSave is the name of theme lookup for document save icon.
	//
	// Since: 2.0
	IconNameDocumentSave fyne.ThemeIconName = "documentSave"

	// IconNameDragCornerIndicator is the name of the icon used in inner windows to
indicate a draggable corner.
	//
	// Since: 2.5
	IconNameDragCornerIndicator fyne.ThemeIconName = "dragCornerIndicator"

	// IconNameMoreHorizontal is the name of theme lookup for horizontal more.
	//
	// Since 2.0
	IconNameMoreHorizontal fyne.ThemeIconName = "moreHorizontal"

	// IconNameMoreVertical is the name of theme lookup for vertical more.
	//
	// Since 2.0
	IconNameMoreVertical fyne.ThemeIconName = "moreVertical"

	// IconNameMailAttachment is the name of theme lookup for mail attachment icon.
	//
	// Since: 2.0
	IconNameMailAttachment fyne.ThemeIconName = "mailAttachment"
```

```
	// IconNameMailCompose is the name of theme lookup for mail compose icon.
	//
	// Since: 2.0
	IconNameMailCompose fyne.ThemeIconName = "mailCompose"

	// IconNameMailForward is the name of theme lookup for mail forward icon.
	//
	// Since: 2.0
	IconNameMailForward fyne.ThemeIconName = "mailForward"

	// IconNameMailReply is the name of theme lookup for mail reply icon.
	//
	// Since: 2.0
	IconNameMailReply fyne.ThemeIconName = "mailReply"

	// IconNameMailReplyAll is the name of theme lookup for mail reply-all icon.
	//
	// Since: 2.0
	IconNameMailReplyAll fyne.ThemeIconName = "mailReplyAll"

	// IconNameMailSend is the name of theme lookup for mail send icon.
	//
	// Since: 2.0
	IconNameMailSend fyne.ThemeIconName = "mailSend"

	// IconNameMediaMusic is the name of theme lookup for media music icon.
	//
	// Since: 2.1
	IconNameMediaMusic fyne.ThemeIconName = "mediaMusic"

	// IconNameMediaPhoto is the name of theme lookup for media photo icon.
	//
	// Since: 2.1
	IconNameMediaPhoto fyne.ThemeIconName = "mediaPhoto"

	// IconNameMediaVideo is the name of theme lookup for media video icon.
	//
	// Since: 2.1
	IconNameMediaVideo fyne.ThemeIconName = "mediaVideo"

	// IconNameMediaFastForward is the name of theme lookup for media fast-forward
icon.
	//
	// Since: 2.0
	IconNameMediaFastForward fyne.ThemeIconName = "mediaFastForward"

	// IconNameMediaFastRewind is the name of theme lookup for media fast-rewind
icon.
	//
	// Since: 2.0
	IconNameMediaFastRewind fyne.ThemeIconName = "mediaFastRewind"

	// IconNameMediaPause is the name of theme lookup for media pause icon.
	//
	// Since: 2.0
	IconNameMediaPause fyne.ThemeIconName = "mediaPause"

	// IconNameMediaPlay is the name of theme lookup for media play icon.
	//
	// Since: 2.0
	IconNameMediaPlay fyne.ThemeIconName = "mediaPlay"

	// IconNameMediaRecord is the name of theme lookup for media record icon.
	//
	// Since: 2.0
	IconNameMediaRecord fyne.ThemeIconName = "mediaRecord"

	// IconNameMediaReplay is the name of theme lookup for media replay icon.
	//
	// Since: 2.0
	IconNameMediaReplay fyne.ThemeIconName = "mediaReplay"
```

```
	// IconNameMediaSkipNext is the name of theme lookup for media skip next icon.
	//
	// Since: 2.0
	IconNameMediaSkipNext fyne.ThemeIconName = "mediaSkipNext"

	// IconNameMediaSkipPrevious is the name of theme lookup for media skip previous
icon.
	//
	// Since: 2.0
	IconNameMediaSkipPrevious fyne.ThemeIconName = "mediaSkipPrevious"

	// IconNameMediaStop is the name of theme lookup for media stop icon.
	//
	// Since: 2.0
	IconNameMediaStop fyne.ThemeIconName = "mediaStop"

	// IconNameMoveDown is the name of theme lookup for move down icon.
	//
	// Since: 2.0
	IconNameMoveDown fyne.ThemeIconName = "arrowDown"

	// IconNameMoveUp is the name of theme lookup for move up icon.
	//
	// Since: 2.0
	IconNameMoveUp fyne.ThemeIconName = "arrowUp"

	// IconNameNavigateBack is the name of theme lookup for navigate back icon.
	//
	// Since: 2.0
	IconNameNavigateBack fyne.ThemeIconName = "arrowBack"

	// IconNameNavigateNext is the name of theme lookup for navigate next icon.
	//
	// Since: 2.0
	IconNameNavigateNext fyne.ThemeIconName = "arrowForward"

	// IconNameArrowDropDown is the name of theme lookup for drop-down arrow icon.
	//
	// Since: 2.0
	IconNameArrowDropDown fyne.ThemeIconName = "arrowDropDown"

	// IconNameArrowDropUp is the name of theme lookup for drop-up arrow icon.
	//
	// Since: 2.0
	IconNameArrowDropUp fyne.ThemeIconName = "arrowDropUp"

	// IconNameFile is the name of theme lookup for file icon.
	//
	// Since: 2.0
	IconNameFile fyne.ThemeIconName = "file"

	// IconNameFileApplication is the name of theme lookup for file application icon.
	//
	// Since: 2.0
	IconNameFileApplication fyne.ThemeIconName = "fileApplication"

	// IconNameFileAudio is the name of theme lookup for file audio icon.
	//
	// Since: 2.0
	IconNameFileAudio fyne.ThemeIconName = "fileAudio"

	// IconNameFileImage is the name of theme lookup for file image icon.
	//
	// Since: 2.0
	IconNameFileImage fyne.ThemeIconName = "fileImage"

	// IconNameFileText is the name of theme lookup for file text icon.
	//
	// Since: 2.0
	IconNameFileText fyne.ThemeIconName = "fileText"
```

```
// IconNameFileVideo is the name of theme lookup for file video icon.
//
// Since: 2.0
IconNameFileVideo fyne.ThemeIconName = "fileVideo"

// IconNameFolder is the name of theme lookup for folder icon.
//
// Since: 2.0
IconNameFolder fyne.ThemeIconName = "folder"

// IconNameFolderNew is the name of theme lookup for folder new icon.
//
// Since: 2.0
IconNameFolderNew fyne.ThemeIconName = "folderNew"

// IconNameFolderOpen is the name of theme lookup for folder open icon.
//
// Since: 2.0
IconNameFolderOpen fyne.ThemeIconName = "folderOpen"

// IconNameHelp is the name of theme lookup for help icon.
//
// Since: 2.0
IconNameHelp fyne.ThemeIconName = "help"

// IconNameHistory is the name of theme lookup for history icon.
//
// Since: 2.0
IconNameHistory fyne.ThemeIconName = "history"

// IconNameHome is the name of theme lookup for home icon.
//
// Since: 2.0
IconNameHome fyne.ThemeIconName = "home"

// IconNameSettings is the name of theme lookup for settings icon.
//
// Since: 2.0
IconNameSettings fyne.ThemeIconName = "settings"

// IconNameStorage is the name of theme lookup for storage icon.
//
// Since: 2.0
IconNameStorage fyne.ThemeIconName = "storage"

// IconNameUpload is the name of theme lookup for upload icon.
//
// Since: 2.0
IconNameUpload fyne.ThemeIconName = "upload"

// IconNameViewFullScreen is the name of theme lookup for view fullscreen icon.
//
// Since: 2.0
IconNameViewFullScreen fyne.ThemeIconName = "viewFullScreen"

// IconNameViewRefresh is the name of theme lookup for view refresh icon.
//
// Since: 2.0
IconNameViewRefresh fyne.ThemeIconName = "viewRefresh"

// IconNameViewZoomFit is the name of theme lookup for view zoom fit icon.
//
// Since: 2.0
IconNameViewZoomFit fyne.ThemeIconName = "viewZoomFit"

// IconNameViewZoomIn is the name of theme lookup for view zoom in icon.
//
// Since: 2.0
IconNameViewZoomIn fyne.ThemeIconName = "viewZoomIn"
```

```
// IconNameViewZoomOut is the name of theme lookup for view zoom out icon.
//
// Since: 2.0
IconNameViewZoomOut fyne.ThemeIconName = "viewZoomOut"

// IconNameViewRestore is the name of theme lookup for view restore icon.
//
// Since: 2.0
IconNameViewRestore fyne.ThemeIconName = "viewRestore"

// IconNameVisibility is the name of theme lookup for visibility icon.
//
// Since: 2.0
IconNameVisibility fyne.ThemeIconName = "visibility"

// IconNameVisibilityOff is the name of theme lookup for invisibility icon.
//
// Since: 2.0
IconNameVisibilityOff fyne.ThemeIconName = "visibilityOff"

// IconNameVolumeDown is the name of theme lookup for volume down icon.
//
// Since: 2.0
IconNameVolumeDown fyne.ThemeIconName = "volumeDown"

// IconNameVolumeMute is the name of theme lookup for volume mute icon.
//
// Since: 2.0
IconNameVolumeMute fyne.ThemeIconName = "volumeMute"

// IconNameVolumeUp is the name of theme lookup for volume up icon.
//
// Since: 2.0
IconNameVolumeUp fyne.ThemeIconName = "volumeUp"

// IconNameDownload is the name of theme lookup for download icon.
//
// Since: 2.0
IconNameDownload fyne.ThemeIconName = "download"

// IconNameComputer is the name of theme lookup for computer icon.
//
// Since: 2.0
IconNameComputer fyne.ThemeIconName = "computer"

// IconNameDesktop is the name of theme lookup for desktop icon.
//
// Since: 2.5
IconNameDesktop fyne.ThemeIconName = "desktop"

// IconNameAccount is the name of theme lookup for account icon.
//
// Since: 2.1
IconNameAccount fyne.ThemeIconName = "account"

// IconNameCalendar is the name of theme lookup for calendar icon.
//
// Since: 2.6
IconNameCalendar fyne.ThemeIconName = "calendar"

// IconNameLogin is the name of theme lookup for login icon.
//
// Since: 2.1
IconNameLogin fyne.ThemeIconName = "login"

// IconNameLogout is the name of theme lookup for logout icon.
//
// Since: 2.1
IconNameLogout fyne.ThemeIconName = "logout"

// IconNameList is the name of theme lookup for list icon.
```

```
        //
        // Since: 2.1
        IconNameList fyne.ThemeIconName = "list"

        // IconNameGrid is the name of theme lookup for grid icon.
        //
        // Since: 2.1
        IconNameGrid fyne.ThemeIconName = "grid"

        // IconNameWindowClose is the name of theme lookup for window close icon.
        //
        // Since: 2.5
        IconNameWindowClose fyne.ThemeIconName = "windowClose"

        // IconNameWindowMaximize is the name of theme lookup for window maximize icon.
        //
        // Since: 2.5
        IconNameWindowMaximize fyne.ThemeIconName = "windowMaximize"

        // IconNameWindowMinimize is the name of theme lookup for window minimize icon.
        //
        // Since: 2.5
        IconNameWindowMinimize fyne.ThemeIconName = "windowMinimize"
)

const (
        // SizeNameCaptionText is the name of theme lookup for helper text size, normally
smaller than regular text size.
        //
        // Since: 2.0
        SizeNameCaptionText fyne.ThemeSizeName = "helperText"

        // SizeNameInlineIcon is the name of theme lookup for inline icons size.
        //
        // Since: 2.0
        SizeNameInlineIcon fyne.ThemeSizeName = "iconInline"

        // SizeNameInnerPadding is the name of theme lookup for internal widget padding
size.
        //
        // Since: 2.3
        SizeNameInnerPadding fyne.ThemeSizeName = "innerPadding"

        // SizeNameLineSpacing is the name of theme lookup for between text line spacing.
        //
        // Since: 2.3
        SizeNameLineSpacing fyne.ThemeSizeName = "lineSpacing"

        // SizeNamePadding is the name of theme lookup for padding size.
        //
        // Since: 2.0
        SizeNamePadding fyne.ThemeSizeName = "padding"

        // SizeNameScrollBar is the name of theme lookup for the scrollbar size.
        //
        // Since: 2.0
        SizeNameScrollBar fyne.ThemeSizeName = "scrollBar"

        // SizeNameScrollBarSmall is the name of theme lookup for the shrunk scrollbar
size.
        //
        // Since: 2.0
        SizeNameScrollBarSmall fyne.ThemeSizeName = "scrollBarSmall"

        // SizeNameSeparatorThickness is the name of theme lookup for the thickness of a
separator.
        //
        // Since: 2.0
        SizeNameSeparatorThickness fyne.ThemeSizeName = "separator"

        // SizeNameText is the name of theme lookup for text size.
```

```
        //
        // Since: 2.0
        SizeNameText fyne.ThemeSizeName = "text"

        // SizeNameHeadingText is the name of theme lookup for text size of a heading.
        //
        // Since: 2.1
        SizeNameHeadingText fyne.ThemeSizeName = "headingText"

        // SizeNameSubHeadingText is the name of theme lookup for text size of a sub-
heading.
        //
        // Since: 2.1
        SizeNameSubHeadingText fyne.ThemeSizeName = "subHeadingText"

        // SizeNameInputBorder is the name of theme lookup for input border size.
        //
        // Since: 2.0
        SizeNameInputBorder fyne.ThemeSizeName = "inputBorder"

        // SizeNameInputRadius is the name of theme lookup for input corner radius.
        //
        // Since: 2.4
        SizeNameInputRadius fyne.ThemeSizeName = "inputRadius"

        // SizeNameSelectionRadius is the name of theme lookup for selection corner
radius.
        //
        // Since: 2.4
        SizeNameSelectionRadius fyne.ThemeSizeName = "selectionRadius"

        // SizeNameScrollBarRadius is the name of theme lookup for the scroll bar corner
radius.
        //
        // Since: 2.5
        SizeNameScrollBarRadius fyne.ThemeSizeName = "scrollBarRadius"

        // SizeNameWindowButtonHeight is the name of the height for an inner window
titleBar button.
        //
        // Since: 2.6
        SizeNameWindowButtonHeight fyne.ThemeSizeName = "windowButtonHeight"

        // SizeNameWindowButtonRadius is the name of the radius for an inner window
titleBar button.
        //
        // Since: 2.6
        SizeNameWindowButtonRadius fyne.ThemeSizeName = "windowButtonRadius"

        // SizeNameWindowButtonIcon is the name of the width of an inner window titleBar
button.
        //
        // Since: 2.6
        SizeNameWindowButtonIcon fyne.ThemeSizeName = "windowButtonIcon"

        // SizeNameWindowTitleBarHeight is the height for inner window titleBars.
        //
        // Since: 2.6
        SizeNameWindowTitleBarHeight fyne.ThemeSizeName = "windowTitleBarHeight"
)

const (
        // VariantDark is the version of a theme that satisfies a user preference for a
dark look.
        //
        // Since: 2.0
        VariantDark = internaltheme.VariantDark

        // VariantLight is the version of a theme that satisfies a user preference for a
light look.
        //
```

```
        // Since: 2.0
        VariantLight = internaltheme.VariantLight
)
```

Keep in mind to add new constants to the tests at test/theme.go.

func AccountIcon

```
func AccountIcon() fyne.Resource
```

AccountIcon returns a resource containing the standard account icon for the current theme

func BackgroundColor

```
func BackgroundColor() color.Color
```

BackgroundColor returns the theme’s background color.

Deprecated: Use Color(theme.ColorNameBackground) instead.

func BrokenImageIcon

```
func BrokenImageIcon() fyne.Resource
```

BrokenImageIcon returns a resource containing an icon to specify a broken or missing image

Since: `2.4`

func ButtonColor

```
func ButtonColor() color.Color
```

ButtonColor returns the theme’s standard button color.

Deprecated: Use Color(theme.ColorNameButton) instead.

func CalendarIcon

```
func CalendarIcon() fyne.Resource
```

CalendarIcon returns a resource containing the standard account icon for the current theme

Since: `2.6`

func CancelIcon

```
func CancelIcon() fyne.Resource
```

CancelIcon returns a resource containing the standard cancel icon for the current theme

func CaptionTextSize

```
func CaptionTextSize() float32
```

CaptionTextSize returns the size for caption text.

func CheckButtonCheckedIcon

```
func CheckButtonCheckedIcon() fyne.Resource
```

CheckButtonCheckedIcon returns a resource containing the standard checkbox checked icon for the current theme

func CheckButtonFillIcon

```
func CheckButtonFillIcon() fyne.Resource
```

CheckButtonFillIcon returns a resource containing the filled checkbox icon for the current theme.

Since: 2.5

func CheckButtonIcon

```
func CheckButtonIcon() fyne.Resource
```

CheckButtonIcon returns a resource containing the standard checkbox icon for the current theme

func Color

```
func Color(name fyne.ThemeColorName) color.Color
```

Color looks up the named colour for current theme and variant.

Since: 2.5

func ColorAchromaticIcon

```
func ColorAchromaticIcon() fyne.Resource
```

ColorAchromaticIcon returns a resource containing the standard achromatic color icon for the current theme

func ColorChromaticIcon

```
func ColorChromaticIcon() fyne.Resource
```

ColorChromaticIcon returns a resource containing the standard chromatic color icon for the current theme

func ColorForWidget

```
func ColorForWidget(name fyne.ThemeColorName, w fyne.Widget) color.Color
```

ColorForWidget looks up the named colour for the requested widget using the current theme and variant. If the widget theme has been overridden that theme will be used.

Since: 2.5

func ColorPaletteIcon

```
func ColorPaletteIcon() fyne.Resource
```

ColorPaletteIcon returns a resource containing the standard color palette icon for the current theme

func ComputerIcon

```
func ComputerIcon() fyne.Resource
```

ComputerIcon returns a resource containing the standard computer icon for the current theme

func ConfirmIcon

```
func ConfirmIcon() fyne.Resource
```

ConfirmIcon returns a resource containing the standard confirm icon for the current theme

func ContentAddIcon

```
func ContentAddIcon() fyne.Resource
```

ContentAddIcon returns a resource containing the standard content add icon for the current theme

func ContentClearIcon

```
func ContentClearIcon() fyne.Resource
```

ContentClearIcon returns a resource containing the standard content clear icon for the current theme

func ContentCopyIcon

```
func ContentCopyIcon() fyne.Resource
```

ContentCopyIcon returns a resource containing the standard content copy icon for the current theme

func ContentCutIcon

```
func ContentCutIcon() fyne.Resource
```

ContentCutIcon returns a resource containing the standard content cut icon for the current theme

func ContentPasteIcon

```
func ContentPasteIcon() fyne.Resource
```

ContentPasteIcon returns a resource containing the standard content paste icon for the current theme

func ContentRedoIcon

```
func ContentRedoIcon() fyne.Resource
```

ContentRedoIcon returns a resource containing the standard content redo icon for the current theme

func ContentRemoveIcon

```
func ContentRemoveIcon() fyne.Resource
```

ContentRemoveIcon returns a resource containing the standard content remove icon for the current theme

func ContentUndoIcon

```
func ContentUndoIcon() fyne.Resource
```

ContentUndoIcon returns a resource containing the standard content undo icon for the current theme

func Current

```
func Current() fyne.Theme
```

Current returns the theme that is currently used for the running application. It looks up based on user preferences and application configuration.

Since: `2.5`

func CurrentForWidget

```
func CurrentForWidget(w fyne.CanvasObject) fyne.Theme
```

CurrentForWidget returns the theme that is currently used for the specified widget. It looks for widget overrides and falls back to the application’s current theme.

Since: `2.5`

func DarkTheme

```
func DarkTheme() fyne.Theme
```

DarkTheme defines the built-in dark theme colors and sizes.

Deprecated: This method ignores user preference and should not be used, it will be removed in v3.0.

If developers want to ignore user preference for theme variant they can set a custom theme.

func DefaultEmojiFont

```
func DefaultEmojiFont() fyne.Resource
```

DefaultEmojiFont returns the font resource for the built-in emoji font. This may return nil if the application was packaged without an emoji font.

Since: 2.4

func DefaultSymbolFont

```
func DefaultSymbolFont() fyne.Resource
```

DefaultSymbolFont returns the font resource for the built-in symbol font.

Since: 2.2

func DefaultTextBoldFont

```
func DefaultTextBoldFont() fyne.Resource
```

DefaultTextBoldFont returns the font resource for the built-in bold font style.

func DefaultTextBoldItalicFont

```
func DefaultTextBoldItalicFont() fyne.Resource
```

DefaultTextBoldItalicFont returns the font resource for the built-in bold and italic font style.

func DefaultTextFont

```
func DefaultTextFont() fyne.Resource
```

DefaultTextFont returns the font resource for the built-in regular font style.

func DefaultTextItalicFont

```
func DefaultTextItalicFont() fyne.Resource
```

DefaultTextItalicFont returns the font resource for the built-in italic font style.

func DefaultTextMonospaceFont

```
func DefaultTextMonospaceFont() fyne.Resource
```

DefaultTextMonospaceFont returns the font resource for the built-in monospace font face.

func DefaultTheme

```
func DefaultTheme() fyne.Theme
```

DefaultTheme returns a built-in theme that can adapt to the user preference of light or dark colors.

Since: 2.0

func DeleteIcon

```
func DeleteIcon() fyne.Resource
```

DeleteIcon returns a resource containing the standard delete icon for the current theme

func DesktopIcon

```
func DesktopIcon() fyne.Resource
```

DesktopIcon returns a resource containing the standard desktop icon for the current theme

func DisabledButtonColor

```
func DisabledButtonColor() color.Color
```

DisabledButtonColor returns the theme’s disabled button color.

Deprecated: Use Color(theme.ColorNameDisabledButton) instead.

func DisabledColor

```
func DisabledColor() color.Color
```

DisabledColor returns the foreground color for a disabled UI element.

Since: 2.0
Deprecated: Use Color(theme.ColorNameDisabled) instead.

func DisabledTextColor

```
func DisabledTextColor() color.Color
```

DisabledTextColor returns the theme’s disabled text color - this is actually the disabled color since 1.4.

Deprecated: Use Color(theme.ColorNameDisabled) instead.

func DocumentCreateIcon

```
func DocumentCreateIcon() fyne.Resource
```

DocumentCreateIcon returns a resource containing the standard document create icon for the current theme

func DocumentIcon

```
func DocumentIcon() fyne.Resource
```

DocumentIcon returns a resource containing the standard document icon for the current theme

func DocumentPrintIcon

```
func DocumentPrintIcon() fyne.Resource
```

DocumentPrintIcon returns a resource containing the standard document print icon for the current theme

func DocumentSaveIcon

```
func DocumentSaveIcon() fyne.Resource
```

DocumentSaveIcon returns a resource containing the standard document save icon for the current theme

func DownloadIcon

```
func DownloadIcon() fyne.Resource
```

DownloadIcon returns a resource containing the standard download icon for the current theme

func ErrorColor

```
func ErrorColor() color.Color
```

ErrorColor returns the theme’s error foreground color.

Since: 2.0
Deprecated: Use Color(theme.ColorNameError) instead.

func ErrorIcon

```
func ErrorIcon() fyne.Resource
```

ErrorIcon returns a resource containing the standard dialog error icon for the current theme

func FileApplicationIcon

```
func FileApplicationIcon() fyne.Resource
```

FileApplicationIcon returns a resource containing the file icon representing application files for the current theme

func FileAudioIcon

```
func FileAudioIcon() fyne.Resource
```

FileAudioIcon returns a resource containing the file icon representing audio files for the current theme

func FileIcon

```
func FileIcon() fyne.Resource
```

FileIcon returns a resource containing the appropriate file icon for the current theme

func FileImageIcon

```
func FileImageIcon() fyne.Resource
```

FileImageIcon returns a resource containing the file icon representing image files for the current theme

func FileTextIcon

```
func FileTextIcon() fyne.Resource
```

FileTextIcon returns a resource containing the file icon representing text files for the current theme

func FileVideoIcon

```
func FileVideoIcon() fyne.Resource
```

FileVideoIcon returns a resource containing the file icon representing video files for the current theme

func FocusColor

```
func FocusColor() color.Color
```

FocusColor returns the color used to highlight a focused widget.

Deprecated: Use Color(theme.ColorNameFocus) instead.

func FolderIcon

```
func FolderIcon() fyne.Resource
```

FolderIcon returns a resource containing the standard folder icon for the current theme

func FolderNewIcon

```
func FolderNewIcon() fyne.Resource
```

FolderNewIcon returns a resource containing the standard folder creation icon for the current theme

func FolderOpenIcon

```
func FolderOpenIcon() fyne.Resource
```

FolderOpenIcon returns a resource containing the standard folder open icon for the current theme

func Font

```
func Font(style fyne.TextStyle) fyne.Resource
```

Font looks up the font for current theme and text style.

Since: 2.5

func ForegroundColor

```
func ForegroundColor() color.Color
```

ForegroundColor returns the theme's standard foreground color for text and icons.

Since: 2.0
Deprecated: Use Color(theme.ColorNameForeground) instead.

func FromJSON

```
func FromJSON(data string) (fyne.Theme, error)
```

FromJSON returns a Theme created from the given JSON metadata. Any values not present in the data will fall back to the default theme. If a parse error occurs it will be returned along with a default theme.

Since: 2.2

func FromJSONReader

```
func FromJSONReader(r io.Reader) (fyne.Theme, error)
```

FromJSONReader returns a Theme created from the given JSON metadata through the reader. Any values not present in the data will fall back to the default theme. If a parse error occurs it will be returned along with a default theme.

Since: 2.2

func FromLegacy

```
func FromLegacy(t fyne.LegacyTheme) fyne.Theme
```

FromLegacy returns a 2.0 Theme created from the given LegacyTheme data. This is a transition path and will be removed in the future (probably version 3.0).

Since: 2.0

func FyneLogo

```
func FyneLogo() fyne.Resource
```

FyneLogo returns a resource containing the Fyne logo.

Deprecated: Applications should use their own icon in most cases.

func GridIcon

```
func GridIcon() fyne.Resource
```

GridIcon returns a resource containing the standard grid icon for the current theme

func HeaderBackgroundColor

```
func HeaderBackgroundColor() color.Color
```

HeaderBackgroundColor returns the color used to draw underneath collection headers.

Since: 2.4
Deprecated: Use Color(theme.ColorNameHeaderBackground) instead.

func HelpIcon

```
func HelpIcon() fyne.Resource
```

HelpIcon returns a resource containing the standard help icon for the current theme

func HistoryIcon

```
func HistoryIcon() fyne.Resource
```

HistoryIcon returns a resource containing the standard history icon for the current theme

func HomeIcon

```
func HomeIcon() fyne.Resource
```

HomeIcon returns a resource containing the standard home folder icon for the current theme

func HoverColor

```
func HoverColor() color.Color
```

HoverColor returns the color used to highlight interactive elements currently under a cursor.

Deprecated: Use Color(theme.ColorNameHover) instead.

func HyperlinkColor

```
func HyperlinkColor() color.Color
```

HyperlinkColor returns the color used for the Hyperlink widget and hyperlink text elements.

Deprecated: Use Color(theme.ColorNameHyperlink) instead.

func Icon

```
func Icon(name fyne.ThemeIconName) fyne.Resource
```

Icon looks up the specified icon for current theme.

Since: `2.5`

func IconForWidget

```
func IconForWidget(name fyne.ThemeIconName, w fyne.Widget) fyne.Resource
```

IconForWidget looks up the specified icon for requested widget using the current theme. If the widget theme has been overridden that theme will be used.

Since: `2.5`

func IconInlineSize

```
func IconInlineSize() float32
```

IconInlineSize is the standard size of icons which appear within buttons, labels etc.

func InfoIcon

```
func InfoIcon() fyne.Resource
```

InfoIcon returns a resource containing the standard dialog info icon for the current theme

func InnerPadding

```
func InnerPadding() float32
```

InnerPadding is the standard gap between element content and the outside edge of a widget.

Since: `2.3`

func InputBackgroundColor

```
func InputBackgroundColor() color.Color
```

InputBackgroundColor returns the color used to draw underneath input elements.

Deprecated: Use Color(theme.ColorNameInputBackground) instead.

func InputBorderColor

```
func InputBorderColor() color.Color
```

InputBorderColor returns the color used to draw underneath input elements.

Since: 2.3
Deprecated: Use Color(theme.ColorNameInputBorder) instead.

func InputBorderSize

```
func InputBorderSize() float32
```

InputBorderSize returns the input border size (or underline size for an entry).

Since: 2.0

func InputRadiusSize

```
func InputRadiusSize() float32
```

InputRadiusSize returns the input radius size.

Since: 2.4

func LightTheme

```
func LightTheme() fyne.Theme
```

LightTheme defines the built-in light theme colors and sizes.

Deprecated: This method ignores user preference and should not be used, it will be removed in v3.0.

If developers want to ignore user preference for theme variant they can set a custom theme.

func LineSpacing

```
func LineSpacing() float32
```

LineSpacing is the default gap between multiple lines of text.

Since: 2.3

func ListIcon

```
func ListIcon() fyne.Resource
```

ListIcon returns a resource containing the standard list icon for the current theme

func LoginIcon

```
func LoginIcon() fyne.Resource
```

LoginIcon returns a resource containing the standard login icon for the current theme

func LogoutIcon

```
func LogoutIcon() fyne.Resource
```

LogoutIcon returns a resource containing the standard logout icon for the current theme

func MailAttachmentIcon

```
func MailAttachmentIcon() fyne.Resource
```

MailAttachmentIcon returns a resource containing the standard mail attachment icon for the current theme

func MailComposeIcon

```
func MailComposeIcon() fyne.Resource
```

MailComposeIcon returns a resource containing the standard mail compose icon for the current theme

func MailForwardIcon

```
func MailForwardIcon() fyne.Resource
```

MailForwardIcon returns a resource containing the standard mail forward icon for the current theme

func MailReplyAllIcon

```
func MailReplyAllIcon() fyne.Resource
```

MailReplyAllIcon returns a resource containing the standard mail reply all icon for the current theme

func MailReplyIcon

```
func MailReplyIcon() fyne.Resource
```

MailReplyIcon returns a resource containing the standard mail reply icon for the current theme

func MailSendIcon

```
func MailSendIcon() fyne.Resource
```

MailSendIcon returns a resource containing the standard mail send icon for the current theme

func MediaFastForwardIcon

```
func MediaFastForwardIcon() fyne.Resource
```

MediaFastForwardIcon returns a resource containing the standard media fast-forward icon for the current theme

func MediaFastRewindIcon

```
func MediaFastRewindIcon() fyne.Resource
```

MediaFastRewindIcon returns a resource containing the standard media fast-rewind icon for the current theme

func MediaMusicIcon

```
func MediaMusicIcon() fyne.Resource
```

MediaMusicIcon returns a resource containing the standard media music icon for the current theme

Since: `2.1`

func MediaPauseIcon

```
func MediaPauseIcon() fyne.Resource
```

MediaPauseIcon returns a resource containing the standard media pause icon for the current theme

func MediaPhotoIcon

```
func MediaPhotoIcon() fyne.Resource
```

MediaPhotoIcon returns a resource containing the standard media photo icon for the current theme

Since: `2.1`

func MediaPlayIcon

```
func MediaPlayIcon() fyne.Resource
```

MediaPlayIcon returns a resource containing the standard media play icon for the current theme

func MediaRecordIcon

```
func MediaRecordIcon() fyne.Resource
```

MediaRecordIcon returns a resource containing the standard media record icon for the current theme

func MediaReplayIcon

```
func MediaReplayIcon() fyne.Resource
```

MediaReplayIcon returns a resource containing the standard media replay icon for the current theme

func MediaSkipNextIcon

```
func MediaSkipNextIcon() fyne.Resource
```

MediaSkipNextIcon returns a resource containing the standard media skip next icon for the current theme

func MediaSkipPreviousIcon

```
func MediaSkipPreviousIcon() fyne.Resource
```

MediaSkipPreviousIcon returns a resource containing the standard media skip previous icon for the current theme

func MediaStopIcon

```
func MediaStopIcon() fyne.Resource
```

MediaStopIcon returns a resource containing the standard media stop icon for the current theme

func MediaVideoIcon

```
func MediaVideoIcon() fyne.Resource
```

MediaVideoIcon returns a resource containing the standard media video icon for the current theme

Since: `2.1`

func MenuBackgroundColor

```
func MenuBackgroundColor() color.Color
```

MenuBackgroundColor returns the theme’s background color for menus.

Since: `2.3`
Deprecated: Use Color(theme.ColorNameMenuBackground) instead.

func MenuDropDownIcon

```
func MenuDropDownIcon() fyne.Resource
```

MenuDropDownIcon returns a resource containing the standard menu drop down icon for the current theme

func MenuDropUpIcon

```
func MenuDropUpIcon() fyne.Resource
```

MenuDropUpIcon returns a resource containing the standard menu drop up icon for the current theme

func MenuExpandIcon

```
func MenuExpandIcon() fyne.Resource
```

MenuExpandIcon returns a resource containing the standard (mobile) expand “submenu icon for the current theme

func MenuIcon

```
func MenuIcon() fyne.Resource
```

MenuIcon returns a resource containing the standard (mobile) menu icon for the current theme

func MoreHorizontalIcon

```
func MoreHorizontalIcon() fyne.Resource
```

MoreHorizontalIcon returns a resource containing the standard horizontal more icon for the current theme

func MoreVerticalIcon

```
func MoreVerticalIcon() fyne.Resource
```

MoreVerticalIcon returns a resource containing the standard vertical more icon for the current theme

func MoveDownIcon

```
func MoveDownIcon() fyne.Resource
```

MoveDownIcon returns a resource containing the standard down arrow icon for the current theme

func MoveUpIcon

```
func MoveUpIcon() fyne.Resource
```

MoveUpIcon returns a resource containing the standard up arrow icon for the current theme

func NavigateBackIcon

```
func NavigateBackIcon() fyne.Resource
```

NavigateBackIcon returns a resource containing the standard backward navigation icon for the current theme

func NavigateNextIcon

```
func NavigateNextIcon() fyne.Resource
```

NavigateNextIcon returns a resource containing the standard forward navigation icon for the current theme

func OverlayBackgroundColor

```
func OverlayBackgroundColor() color.Color
```

OverlayBackgroundColor returns the theme’s background color for overlays like dialogs.

Since: `2.3`
Deprecated: Use Color(theme.ColorNameOverlayBackground) instead.

func Padding

```
func Padding() float32
```

Padding is the standard gap between elements and the border around interface elements.

func PlaceHolderColor

```
func PlaceHolderColor() color.Color
```

PlaceHolderColor returns the theme’s standard text color.

Deprecated: Use Color(theme.ColorNamePlaceHolder) instead.

func PressedColor

```
func PressedColor() color.Color
```

PressedColor returns the color used to overlap tapped features.

Since: 2.0
Deprecated: Use Color(theme.ColorNamePressed) instead.

func PrimaryColor

```
func PrimaryColor() color.Color
```

PrimaryColor returns the color used to highlight primary features.

Deprecated: Use Color(theme.ColorNamePrimary) instead.

func PrimaryColorNamed

```
func PrimaryColorNamed(name string) color.Color
```

PrimaryColorNamed returns a theme specific color value for a named primary color.

Since: 1.4
Deprecated: You should not access named primary colors but access the primary color using Color(theme.ColorNamePrimary) instead.

func PrimaryColorNames

```
func PrimaryColorNames() []string
```

PrimaryColorNames returns a list of the standard primary color options.

Since: 1.4

func QuestionIcon

```
func QuestionIcon() fyne.Resource
```

QuestionIcon returns a resource containing the standard dialog question icon for the current theme

func RadioButtonCheckedIcon

```
func RadioButtonCheckedIcon() fyne.Resource
```

RadioButtonCheckedIcon returns a resource containing the standard radio button checked icon for the current theme

func RadioButtonFillIcon

```
func RadioButtonFillIcon() fyne.Resource
```

RadioButtonFillIcon returns a resource containing the filled checkbox icon for the current theme.

Since: 2.5

func RadioButtonIcon

```
func RadioButtonIcon() fyne.Resource
```

RadioButtonIcon returns a resource containing the standard radio button icon for the current theme

func ScrollBarColor

```
func ScrollBarColor() color.Color
```

ScrollBarColor returns the color (and translucency) for a scrollBar.

Deprecated: Use Color(theme.ColorNameScrollBar) instead.

func ScrollBarSize

```
func ScrollBarSize() float32
```

ScrollBarSize is the width (or height) of the bars on a ScrollContainer.

func ScrollBarSmallSize

```
func ScrollBarSmallSize() float32
```

ScrollBarSmallSize is the width (or height) of the minimized bars on a ScrollContainer.

func SearchIcon

```
func SearchIcon() fyne.Resource
```

SearchIcon returns a resource containing the standard search icon for the current theme

func SearchReplaceIcon

```
func SearchReplaceIcon() fyne.Resource
```

SearchReplaceIcon returns a resource containing the standard search and replace icon for the current theme

func SelectionColor

```
func SelectionColor() color.Color
```

SelectionColor returns the color for a selected element.

Since: `2.1`
Deprecated: Use Color(theme.ColorNameSelection) instead.

func SelectionRadiusSize

```
func SelectionRadiusSize() float32
```

SelectionRadiusSize returns the selection highlight radius size.

Since: `2.4`

func SeparatorColor

```
func SeparatorColor() color.Color
```

SeparatorColor returns the color for the separator element.

Since: `2.3`
Deprecated: Use Color(theme.ColorNameSeparator) instead.

func SeparatorThicknessSize

```
func SeparatorThicknessSize() float32
```

SeparatorThicknessSize is the standard thickness of the separator widget.

Since: `2.0`

func SettingsIcon

```
func SettingsIcon() fyne.Resource
```

SettingsIcon returns a resource containing the standard settings icon for the current theme

func ShadowColor

```
func ShadowColor() color.Color
```

ShadowColor returns the color (and translucency) for shadows used for indicating elevation.

Deprecated: Use Color(theme.ColorNameShadow) instead.

func Size

```
func Size(name fyne.ThemeSizeName) float32
```

Size looks up the specified size for current theme.

Since: 2.5

func SizeForWidget

```
func SizeForWidget(name fyne.ThemeSizeName, w fyne.Widget) float32
```

SizeForWidget looks up the specified size for the requested widget using the current theme. If the widget theme has been overridden that theme will be used.

Since: 2.5

func StorageIcon

```
func StorageIcon() fyne.Resource
```

StorageIcon returns a resource containing the standard storage icon for the current theme

func SuccessColor

```
func SuccessColor() color.Color
```

SuccessColor returns the theme’s success foreground color.

Since: 2.3
Deprecated: Use Color(theme.ColorNameSuccess) instead.

func SymbolFont

```
func SymbolFont() fyne.Resource
```

SymbolFont returns the font resource for the symbol font style.

Since: 2.4

func TextBoldFont

```
func TextBoldFont() fyne.Resource
```

TextBoldFont returns the font resource for the bold font style.

func TextBoldItalicFont

```
func TextBoldItalicFont() fyne.Resource
```

TextBoldItalicFont returns the font resource for the bold and italic font style.

func TextColor

```
func TextColor() color.Color
```

TextColor returns the theme’s standard text color - this is actually the foreground color since 1.4.

Deprecated: Use theme.ForegroundColor() colour instead.

func TextFont

```
func TextFont() fyne.Resource
```

TextFont returns the font resource for the regular font style.

func TextHeadingSize

```
func TextHeadingSize() float32
```

TextHeadingSize returns the text size for header text.

Since: `2.1`

func TextItalicFont

```
func TextItalicFont() fyne.Resource
```

TextItalicFont returns the font resource for the italic font style.

func TextMonospaceFont

```
func TextMonospaceFont() fyne.Resource
```

TextMonospaceFont returns the font resource for the monospace font face.

func TextSize

```
func TextSize() float32
```

TextSize returns the standard text size.

func TextSubHeadingSize

```
func TextSubHeadingSize() float32
```

TextSubHeadingSize returns the text size for sub-header text.

Since: `2.1`

func UploadIcon

```
func UploadIcon() fyne.Resource
```

UploadIcon returns a resource containing the standard upload icon for the current theme

func ViewFullScreenIcon

```
func ViewFullScreenIcon() fyne.Resource
```

ViewFullScreenIcon returns a resource containing the standard fullscreen icon for the current theme

func ViewRefreshIcon

```
func ViewRefreshIcon() fyne.Resource
```

ViewRefreshIcon returns a resource containing the standard refresh icon for the current theme

func ViewRestoreIcon

```
func ViewRestoreIcon() fyne.Resource
```

ViewRestoreIcon returns a resource containing the standard exit fullscreen icon for the current theme

func VisibilityIcon

```
func VisibilityIcon() fyne.Resource
```

VisibilityIcon returns a resource containing the standard visibility icon for the current theme

func VisibilityOffIcon

```
func VisibilityOffIcon() fyne.Resource
```

VisibilityOffIcon returns a resource containing the standard visibility off icon for the current theme

func VolumeDownIcon

```
func VolumeDownIcon() fyne.Resource
```

VolumeDownIcon returns a resource containing the standard volume down icon for the current theme

func VolumeMuteIcon

```
func VolumeMuteIcon() fyne.Resource
```

VolumeMuteIcon returns a resource containing the standard volume mute icon for the current theme

func VolumeUpIcon

```
func VolumeUpIcon() fyne.Resource
```

VolumeUpIcon returns a resource containing the standard volume up icon for the current theme

func WarningColor

```
func WarningColor() color.Color
```

WarningColor returns the theme's warning foreground color.

Since: `2.3`
Deprecated: Use Color(theme.ColorNameWarning) instead.

func WarningIcon

```
func WarningIcon() fyne.Resource
```

WarningIcon returns a resource containing the standard dialog warning icon for the current theme

func WindowCloseIcon

```
func WindowCloseIcon() fyne.Resource
```

WindowCloseIcon returns a resource containing the window close icon for the current theme

Since: `2.5`

func WindowMaximizeIcon

```
func WindowMaximizeIcon() fyne.Resource
```

WindowMaximizeIcon returns a resource containing the window maximize icon for the current theme

Since: `2.5`

func WindowMinimizeIcon

```
func WindowMinimizeIcon() fyne.Resource
```

WindowMinimizeIcon returns a resource containing the window minimize icon for the current theme

Since: 2.5

func ZoomFitIcon

```
func ZoomFitIcon() fyne.Resource
```

ZoomFitIcon returns a resource containing the standard zoom fit icon for the current theme

func ZoomInIcon

```
func ZoomInIcon() fyne.Resource
```

ZoomInIcon returns a resource containing the standard zoom in icon for the current theme

func ZoomOutIcon

```
func ZoomOutIcon() fyne.Resource
```

ZoomOutIcon returns a resource containing the standard zoom out icon for the current theme

types

- DisabledResource
- ErrorThemedResource
- InvertedThemedResource
- PrimaryThemedResource
- ThemedResource

Fyne API "widget"

widget

```
import "fyne.io/fyne/v2/widget"
```

Package widget defines the UI widgets within the Fyne toolkit.

Usage

```
var (
        // RichTextStyleBlockquote represents a quote presented in an indented block.
        //
        // Since: 2.1
        RichTextStyleBlockquote = RichTextStyle{
                ColorName: theme.ColorNameForeground,
                Inline:    false,
                SizeName:  theme.SizeNameText,
                TextStyle: fyne.TextStyle{Italic: true},
        }
        // RichTextStyleCodeBlock represents a code blog segment.
        //
        // Since: 2.1
        RichTextStyleCodeBlock = RichTextStyle{
                ColorName: theme.ColorNameForeground,
                Inline:    false,
                SizeName:  theme.SizeNameText,
                TextStyle: fyne.TextStyle{Monospace: true},
        }
        // RichTextStyleCodeInline represents an inline code segment.
```

```
	//
	// Since: 2.1
	RichTextStyleCodeInline = RichTextStyle{
		ColorName: theme.ColorNameForeground,
		Inline:    true,
		SizeName:  theme.SizeNameText,
		TextStyle: fyne.TextStyle{Monospace: true},
	}
	// RichTextStyleEmphasis represents regular text with emphasis.
	//
	// Since: 2.1
	RichTextStyleEmphasis = RichTextStyle{
		ColorName: theme.ColorNameForeground,
		Inline:    true,
		SizeName:  theme.SizeNameText,
		TextStyle: fyne.TextStyle{Italic: true},
	}
	// RichTextStyleHeading represents a heading text that stands on its own line.
	//
	// Since: 2.1
	RichTextStyleHeading = RichTextStyle{
		ColorName: theme.ColorNameForeground,
		Inline:    false,
		SizeName:  theme.SizeNameHeadingText,
		TextStyle: fyne.TextStyle{Bold: true},
	}
	// RichTextStyleInline represents standard text that can be surrounded by other
elements.
	//
	// Since: 2.1
	RichTextStyleInline = RichTextStyle{
		ColorName: theme.ColorNameForeground,
		Inline:    true,
		SizeName:  theme.SizeNameText,
	}
	// RichTextStyleParagraph represents standard text that should appear separate
from other text.
	//
	// Since: 2.1
	RichTextStyleParagraph = RichTextStyle{
		ColorName: theme.ColorNameForeground,
		Inline:    false,
		SizeName:  theme.SizeNameText,
	}
	// RichTextStylePassword represents standard sized text where the characters are
obscured.
	//
	// Since: 2.1
	RichTextStylePassword = RichTextStyle{
		ColorName: theme.ColorNameForeground,
		Inline:    true,
		SizeName:  theme.SizeNameText,
	}
	// RichTextStyleStrong represents regular text with a strong emphasis.
	//
	// Since: 2.1
	RichTextStyleStrong = RichTextStyle{
		ColorName: theme.ColorNameForeground,
		Inline:    true,
		SizeName:  theme.SizeNameText,
		TextStyle: fyne.TextStyle{Bold: true},
	}
	// RichTextStyleSubHeading represents a sub-heading text that stands on its own
line.
	//
	// Since: 2.1
	RichTextStyleSubHeading = RichTextStyle{
		ColorName: theme.ColorNameForeground,
		Inline:    false,
		SizeName:  theme.SizeNameSubHeadingText,
		TextStyle: fyne.TextStyle{Bold: true},
```

```
        }
)
```

func NewSimpleRenderer

```
func NewSimpleRenderer(object fyne.CanvasObject) fyne.WidgetRenderer
```

NewSimpleRenderer creates a new SimpleRenderer to render a widget using a single fyne.CanvasObject.

Since: `2.1`

func ShowModalPopUp

```
func ShowModalPopUp(content fyne.CanvasObject, canvas fyne.Canvas)
```

ShowModalPopUp creates a new popUp for the specified content and displays it on the passed canvas. A modal PopUp blocks interactions with underlying elements, covered with a semi-transparent overlay.

func ShowPopUp

```
func ShowPopUp(content fyne.CanvasObject, canvas fyne.Canvas)
```

ShowPopUp creates a new popUp for the specified content and displays it on the passed canvas.

func ShowPopUpAtPosition

```
func ShowPopUpAtPosition(content fyne.CanvasObject, canvas fyne.Canvas, pos
fyne.Position)
```

ShowPopUpAtPosition creates a new popUp for the specified content at the specified absolute position. It will then display the popup on the passed canvas.

func ShowPopUpAtRelativePosition

```
func ShowPopUpAtRelativePosition(content fyne.CanvasObject, canvas fyne.Canvas, rel
fyne.Position, to fyne.CanvasObject)
```

ShowPopUpAtRelativePosition shows a new popUp for the specified content at the given position relative to stated object. It will then display the popup on the passed canvas.

Since 2.4

func ShowPopUpMenuAtPosition

```
func ShowPopUpMenuAtPosition(menu *fyne.Menu, c fyne.Canvas, pos fyne.Position)
```

ShowPopUpMenuAtPosition creates a PopUp menu populated with items from the passed menu structure. It will automatically be positioned at the provided location and shown as an overlay on the specified canvas.

func ShowPopUpMenuAtRelativePosition

```
func ShowPopUpMenuAtRelativePosition(menu *fyne.Menu, c fyne.Canvas, rel fyne.Position,
to fyne.CanvasObject)
```

ShowPopUpMenuAtRelativePosition creates a PopUp menu populated with menu items from the passed menu structure. It will automatically be positioned at the given position relative to stated object and shown as an overlay on the specified canvas.

Since 2.4

types

- Accordion
- AccordionItem
- Activity
- BaseWidget
- Button

- ButtonAlign
- ButtonIconPlacement
- ButtonImportance
- ButtonStyle
- Calendar
- Card
- Check
- CheckGroup
- CustomTextGridStyle
- DateEntry
- DisableableWidget
- Entry
- FileIcon
- Form
- FormItem
- GridWrap
- GridWrapItemID
- Hyperlink
- HyperlinkSegment
- Icon
- ImageSegment
- Importance
- Label
- List
- ListItemID
- ListSegment
- Menu
- Orientation
- ParagraphSegment
- PopUp
- PopUpMenu
- ProgressBar
- ProgressBarInfinite
- RadioGroup
- RichText
- RichTextBlock
- RichTextSegment
- RichTextStyle
- Select
- SelectEntry
- Separator
- SeparatorSegment
- Slider
- Table
- TableCellID
- TextGrid
- TextGridCell
- TextGridRow
- TextGridStyle
- TextSegment
- Toolbar
- ToolbarAction
- ToolbarItem
- ToolbarSeparator
- ToolbarSpacer
- Tree
- TreeNodeID